Contents

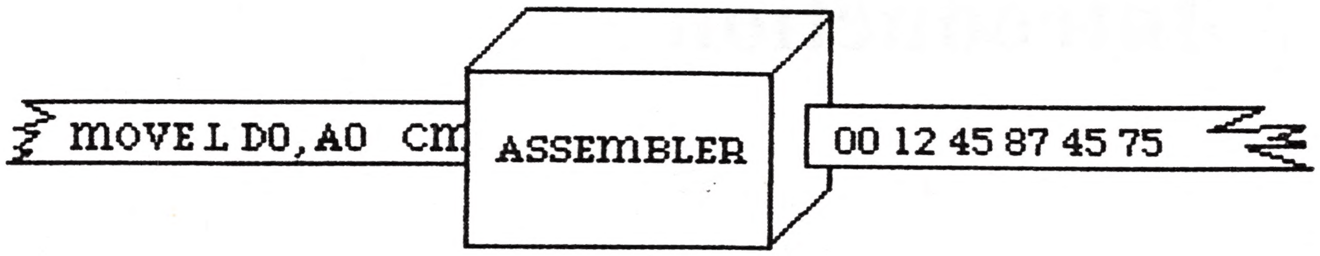
Chapter One
Introduction

This book is designed as a teaching guide to 68000 assembly language. The Motorola 68000 microprocessor is one of a series of three similar processors, the 68008, the 68000, and the 68010. The differences between these are explained Appendix 2, but for programming use, the instructions in this book may be applied to all three.

The 68000 is the main processor in many computers, including the Apple Macintosh, the Commodore Amiga, the Atari 520ST and the Hewlett-Packard Integral PC. It is also the processor in the Apple LaserWriter. The 68008 is the processor used in the Sinclair QL.

Assembly language is one of many computer languages, such as BASIC, FORTRAN, PASCAL etc. It is a low-level language, which means that each instruction in assembly language corresponds to one machine code instruction. High-level languages, such as BASIC, have instructions which each correspond to a routine of several machine instructions. Assembly language is a compiled language, and is completely translated into machine code before the program is run. When the program is run it is this machine code which is used, and the programs are executed very quickly. Once the source program has been compiled into machine code, the machine code can be saved as a binary file on disk. The source program is no longer required to execute the program, and the machine code program is used instead.

Assembly language was developed as an alternative to machine code programming which had the same flexibility as machine code, but which removed the need to represent instructions by numbers, a practice which is not only tedious, but also prone to errors. A language was thus developed to replace the numbers by mnemonics, and which was 'assembled' into machine code by a translator program. This translator program was thus known as an assembler, and the language in which the programs were written became known as assembly language.



Assembly language instructions are called mnemonics (memory aids) because they are intended to indicate the function of their equivalent machine code instruction. For example, *BSR* stands for *Branch to SubRoutine*.

As a result of using mnemonics, programs are much easier to read and understand than if they were in machine code. Being easier to read, they are also far more easily corrected ('debugged').

Instructions are short words related to their function, and are generally followed by one or more items of data ('operands') required for the operation to be performed.

Assemblers are available for all major computers, and now completely eliminate the need for home computers users to use machine code directly when developing programs. In fact, machine code is now generally only used by professional programmers for creating highly specialized applications, such as the fundamental programs of a computer (the operating system).

Since assembly language is an alternative to machine code, it is most frequently used for applications which require the speed and flexibility of machine code, and its use today in the home computer area is often in the development of games, since it can be used to produce more complex scenarios without losing the speed of the game. For business and industry, assembly language is used to create programs which can process large amounts of data in less time than would be taken by a high-level language, and also to produce more sophisticated programs (in terms of program flexibility and capability).

The most valuable use of assembly language is in developing applications for computers, such as communications programs for networks of computers and other utilities which increase the range of functions available to the programmer, including 'patches', small sections of machine code which may be incorporated into computer languages to provide specialized routines to take advantages of the particular abilities of the computer.

Assembly language may be used for these applications because it operates at machine level, i.e. it can directly manipulate areas of the computer in any way which the programmer wishes, without being constrained by the instructions, as often occurrs with many other languages.

In this way, assembly language is one of the most powerful tools a programmer can have in his possession since it enables him to perform virtually any task, limited only by his imagination. Computers are not usually as limited as many people consider them to be - it requires only practice and imagination to produce effects which would often be considered unobtainable on a particular computer!

By copying the examples in the text, fluency in assembly language can be acquired more quickly than by simply reading the text. Use the examples as a guide, a flexible framework, on which to base your own experimentations with the instructions and techniques. Practice is the best way to master any computer language, so that the basics are thoroughly established and easily recalled. Practice is also the best way to learn the eccentricities of your particular assembler program. Each assembler - the translator program which turns your programs into machine code - is different, and each has its own additional set of instructions to take advantage of the features of the computer for which it is written, and also to enhance the standard set of 68000 instructions. These additional instructions include instructions to create labels, to specify the address at which the program code is to be stored, to list an assembly language program, and many others. Practice and (initially) frequent reference to both your assembler manual and this book will soon establish a good working knowledge of 68000 assembly language, and will also familiarise you with your computer. Working at your own pace, moving to a more advanced section only when the basics are thoroughly understood, will increase your knowledge at a steady rate, in a logical sequence.

Chapter Two
Numbers and Data

The memory of a computer can be considered as a series of numbered slots in which information is placed - like the pigeon-hole system used by the Post Office. The number assigned to a location is called its address, and uniquely identifies a particular location in the same way that a house address uniquely identifies a particular house.

Each slot is called a byte, and consists of eight smaller sections, called bits. Bits can have one of two values, either 1 or 0. A bit with the value 1 is said to be 'set'. If it has the value 0, it is said to be 'clear'. Numbers are built up as strings of bits. This system of representing numbers is called the binary system.

The 68000 microprocessor is known as a 16/32 bit processor because it can receive and send up to 16 bits of data at the same time, but it has the capacity to access addresses up to 32 bits long, although in practice only addresses up to 24 bits long can be used, since the processor ignores the other 8 bits.

The central processing unit (CPU) in the processor is the 'brain' of the computer which oversees all operations in the computer.

The CPU consists of various parts :

(a) The control unit - This supervises the passage of data through the CPU.

(b) The arithmetic and logic unit (ALU) - This carries out all the mathematical operations of the computer.

(c) The program counter - This always contains the address of the next instruction to be executed.

(d) Data registers - These store data immediately required for the operations.

(e) Address registers - These hold the addresses of memory locations to which data is being sent ('written to'), or from which data is being obtained ('read from').

(f) Other registers - These are registers which hold 'flags' which hold information about the result of the instruction being performed, and their contents may be used to affect the flow of a program, i.e. a branch to a subroutine in the program will only occur if a particular flag is set - i.e. it has a value of 1.

(g) Buses - These are 'tracks' on which data passes to and from the CPU. There are two buses, the address bus and the data bus. The address bus is 32 bits long, and is used to access memory locations. The address of the memory location being accessed is stored in the address bus, and the data being stored in memory or retrieved from memory is sent to and from the CPU via the data bus.

The CPU organizes the execution of a program in a fetch-execute cycle which continues through the execution of the program.

The fetch-execute cycle has three main steps.

1. Fetch instruction from address held in program counter, and increase the value in the program counter to be the address of the next instruction in memory.

2. Execute instruction.

3. Go back to step (1).

This is continued until an 'end program' instruction is encountered.
Consider, for example, the following series of instructions as a program.

Instruction	**Operation**
1	Place contents of location $1004 into register 1. (Note: location contents remain unaffected)
2	Add contents of location $1005 to contents of register 1.
3	Store result in the location $1004.
4	Stop.

Once this program has been coded into memory, starting at, say, location $1000, the computer memory now looks as follows.

Location	Contents
$1000	1231004 = Instruction 1
$1001	4561005 = Instruction 2
$1002	7891004 = Instruction 3
$1003	000 = Stop
$1004	111 = Data
$1005	222 = Data

Note that computer memory contents cannot normally be listed without a special program called a monitor.

The program counter (PC) is now set to $1000, and the CPU fetches the first instruction from $1000, and increases the PC to $1001. The instruction is carried out, and the contents of register 1 become 111. The CPU fetches the next instruction, and increments the PC. The instruction is executed and the contents of register 1 now become 333. The next instruction is now fetched as before, and the contents of $1004 are changed from 111 to 333. The next instruction is STOP, so the processor now stops fetching instructions from memory.

The computer memory would now look like this :

Location	Contents
1000	1231004 = Instruction 1
1001	4561005 = Instruction 2
1002	7891004 = Instruction 3
1003	000 = Stop
1004	333 = Data
1005	222 = Data

Note that only location $1004 is affected by the program, all other locations remain unchanged.

The use of assembly language means that the programmer does not have to manually place the codes into memory. Instead, he specifies a starting address, and the assembler translates the program into code, starting at that address. Thus, in assembly language the above program is written in a form similar to the following, with instructions which are much more meaningful than the numbers above. Note : this is not a complete machine code program, and will not run as it is given here.

Line No.	Assembly language	Comments
100	MOVE.B $1004, D1	(Load loc.1004 into Reg.1)
200	ADD.B $1005, D1	(Add loc.1005 to Reg.1)
300	MOVE.B D1, $1004	(Store Reg.1 in 1004)
400	END	

As in some other languages, assembly language uses line numbers, and most programs are written with one instruction to a line, for ease of correction.

The .B after each instruction implies that a byte-length data item is being dealt with.

This program is far more readable than its machine code equivalent, and can be listed on-screen like programs in , for example, BASIC. Thus it is far easier to alter or extend. If, for instance, the program has to be extended to add three numbers together, altering the machine code program would involve replacing the contents of virtually every address, which is a very tedious process. Assembly language can be listed, like a BASIC program, and lines can be added and changed easily, without having to completely change each line.

Line No.	Assembly Language	Address	Contents
100	MOVE.B $1005, D1	$1000	1231005
200	ADD.B $1006, D1	$1001	4561006
300	ADD.B $1007, D1	$1002	3211007
400	MOVE.B D1, $1005	$1003	7891005
500	END	$1004	000
		$1005	111
		$1006	222
		$1007	333

Only one line has been added to the assembly language program, whereas six addresses in the machine code program have been moved.

The assembler automatically takes care of these changes when translating the program, making it far easier for a programmer to alter a program. Just imagine adding only one instruction to a large section of machine code, without an assembler!

Instructions and data are represented in memory in binary form, as a string of bits. As each location, or byte, is only eight bits, the largest number which can be held in any location is 11111111 (binary) = 255 (decimal). If numbers larger than this are to be stored, combinations of bytes must be used. The most common combinations are :

byte (8 bits),
word (16 bits=2 bytes)
and *longword* (32 bits = 4 bytes).

So a number larger than 255 is stored as a word, and if it is larger than 65535 it is stored as a longword. However, large numbers are unwieldy to work with in binary, so they are normally written in base sixteen - hexadecimal. Hexadecimal (hex) numbers are distinguished from decimal numbers with a $ (dollar) sign in front of the number. Data numbers are distinguished from address numbers by a # (hash) sign in front of the number.

e.g #1234 is the decimal number 1234

#$1234 is the hex number 1234 = 4660 in decimal

$1234 is an address in memory. By convention, addresses are written in hexadecimal.

Numbers larger than one byte are stored in memory with the *high-order* bits in the *lowest* address, and the *low-order* byte at the *highest* address. The high-order bits are also called the *most-significant* bits, and are the *leftmost* bits of the number. The low-order bits (*least-significant* bits) are the *rightmost* bits of the number. To store the number, the computer splits it into bytes, and then stores them in memory. This 'number-splitting' is done automatically by the processor, without instruction from a programmer.

For instance, the longword number #$00125634 would be split 00 12 56 34. To store it starting at address $1003, the computer would store it as below.

Address	$1003	contains	00
	$1004	"	12
	$1005	"	56
	$1006	"	34

To read the number, the computer is instructed to read a longword number starting at address $1003. The computer reads all four bytes of the longword without further instruction from the programmer.

Numbers larger than 32 bits are also stored in bytes, but the computer cannot access them as one data item, and operations on these numbers have to be performed first on each complete low-order longword, and repeated for the remaining high-order data, which may be byte, word or longword.

Hexadecimal (hex) is chosen, rather than decimal, because conversion from hexadecimal to binary, and vice versa, is far simpler than conversion from decimal to binary. When converting hex to binary, each hex digit represents a nybble (4 bits). Single hex digits have values between 0 and 15. Digits larger than 9 are represented by the alphabetic letters A to F.

Thus A=10, B=11, C=12, D=13, E=14, and F=15

Therefore, to convert a hex number to binary, each hex digit is taken in turn and converted to a nybble.

e.g #$E7D = 1110 0111 1101

To convert from binary to hex, the binary number is first divided into groups of four, *from the right,* and then converting each group into a hex digit. Since the conversion is simple, and each particular hex digit is always represented by the same nybble, hex data can be converted to decimal using a conversion table to determine the value of each nybble of the number. In contrast, conversion of decimal to binary and vice versa involves considerable arithmetic calculation.

The following table may be used for hex / binary conversion.

Hex value	**Decimal value**	**Binary value**
0	**0**	**0000**
1	**1**	**0001**
2	**2**	**0010**
3	**3**	**0011**
4	**4**	**0100**
5	**5**	**0101**
6	**6**	**0110**
7	**7**	**0111**
8	**8**	**1000**
9	**9**	**1001**
A	**10**	**1010**
B	**11**	**1011**
C	**12**	**1100**
D	**13**	**1101**
E	**14**	**1110**
F	**15**	**1111**

The computer performs several operations on binary numbers, and an explanation of these is given below. All binary operations in arithmetic are handled by the ALU, but an understanding of binary numbers and the way in which a computer uses them is very important when programming in assembly language, since it clarifies the way in which certain instructions, such as comparison instructions, operate. It also becomes easier to anticipate how many bytes are going to used in executing operations or storing data, producing programs which are more efficient and compact and contain less errors.

Ones complement

This is the simplest operation performed on binary numbers. Each digit of the binary number is subtracted from 1. A fast way of doing this is to simply replace 1 by 0, and 0 by 1.

```
e.g. the ones complement of      00010110
                         is      11101001.
```

The ones complement may also be called the *diminished complement* of a binary number.

Twos complement

To find the twos complement (also called the *true complement* of a binary number), calculate the ones complement of the number and then add one.

```
e.g. The twos complement of      00010110
                         is      11101001    (ones complement)
                                 _______1+
                          =      11101010
```

Thus the twos complement of 00010110 is 11101010.

Negative numbers

Negative numbers are stored as the twos complement of their absolute(or positive) value.

Positive numbers are stored in pure binary, and the difference between positive and negative is determined by the leftmost (most significant) bit of the number. In signed integer arithmetic, if the number is positive the leftmost bit is *always* 0, and if the number is negative, the leftmost bit is *always* 1.

Since the 68000 always stores negative numbers in twos complement form, the leftmost bit of the data item is reserved as a *sign-bit*. Thus the range of positive numbers which can be represented as one data item is from $0000001 to $7FFFFFFF (in hexadecimal), i.e. 1 to 2147483647 (decimal), and the range of negative numbers is $FFFFFFFF to $80000001 (hex), i.e. -1 to -2147483647 (decimal). Zero is neither positive or negative. Thus, storing -1 (decimal) and storing $FF (hex) as a byte value is the same thing in *signed* arithmetic - the distinction between the two is made only when *unsigned* arithmetic is being used, in which case, $FF (hex) is treated as 255 (decimal). In unsigned arithmetic, the most-significant bit of the number is not considered to be a sign-bit, but is included as part of the entire number.

Addition

Addition is done in the same way as decimal addition, but remember that all the digits in a binary number are either 1 or 0. Thus, adding two 1's together gives the result 0, with a carry to the next column.

```
e.g.        00111011
            00010111+
            01010010
```

If more than two numbers are being added, they are added in groups. i.e. The first two numbers are added together, then the third is added to the result, and the fourth is added to that result, etc.

Subtraction

Subtraction is done by addition! That statement is not as ridiculous as it sounds, since binary subtraction of two numbers is done by adding the first number and the twos complement of the number to be subtracted. The subtraction works because adding a negative number gives the same result as subtracting its positive value.

e.g By normal subtraction :

```
            00010110
            00001101-
            00001001
```

By twos complement subtraction :

Twos complement of 00001101 = 11110010 + 1 = 11110011

```
             00010110
             11110011+
            10000 1001
```

The 1 at the far left indicates the answer is positive and is not included in the final answer. Thus the answer is positive and is 00001101. If the answer were negative, the digit at the far left would be 0.

Since binary addition is much simpler than subtraction, the twos complement method is virtually always used for subtraction. The computer automatically performs the twos complement method of subtraction on execution of a subtraction command.

Logical Operations

AND

When an AND operation is performed on two binary numbers, the result is a binary number *with a 1* in each bit position *where both of the original numbers had a 1*. All other bits contain 0.

e.g. 00010001 AND 00111000 = 00010000

The AND operation is usually mainly to find the remainder of a division by a power of two (2, 4, 8, 16....). To find the remainder, the number being divided is ANDed with the *power of two minus one*.

e.g. For the remainder of 00010001 (17) divided by
00000100 (4), 00010001 (17) is ANDed with 00000011 (3)
00010001 AND 00000011 = 00000001
Thus, the remainder of 17 / 4 is 1.

OR

The OR operation performed on two binary numbers results in a binary number *with a 1* in any bit position *where either or both of the original numbers had a 1*. All other bits are 0.

e.g 00010010 OR 01000101 = 01010111

XOR

The eXclusive OR operation takes two binary numbers and produces a third binary number *with a 1* where *either, but not both, of the original numbers had a 1*. All other bits are 0.

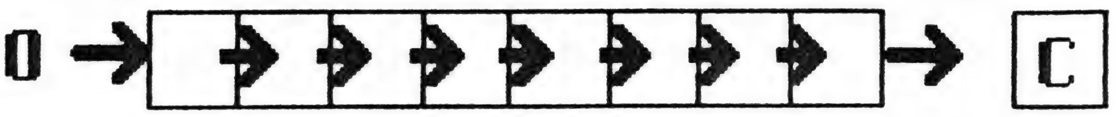
```
e.g.        10001011  XOR  01110100 =  11111111,
but         11001011  XOR  01110100 =  10111111
```

Note : XORing a number with another, all 1, (e.g. 1111) number produces the ones complement of the number. XORing a number with itself gives zero.

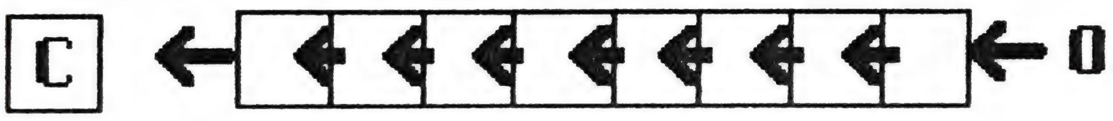
Two other operations may also be performed on binary numbers - shifts and rotates. These may be performed either to the left or to the right. Shifts may be arithmetical or logical.

When a byte is logic shifted, all the bits are moved one position in the direction of the shift. The vacant bit has a zero placed in it. The bit at the end of the byte 'falls off' into the carry flag, a special bit in the status register of the CPU. This flag receives the bit which would otherwise be lost.

Logical Right Shift



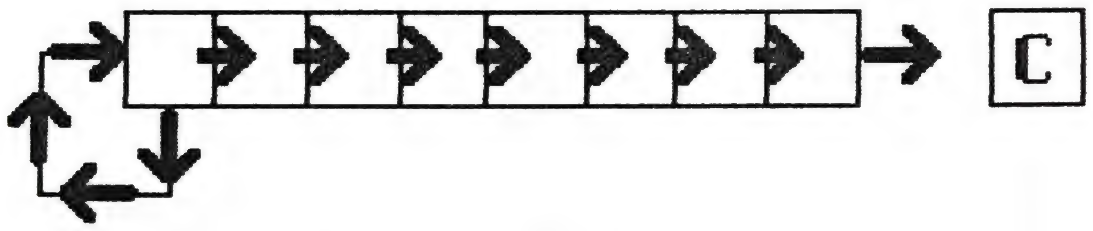
Logical Left Shift & Arithmetic Left Shift



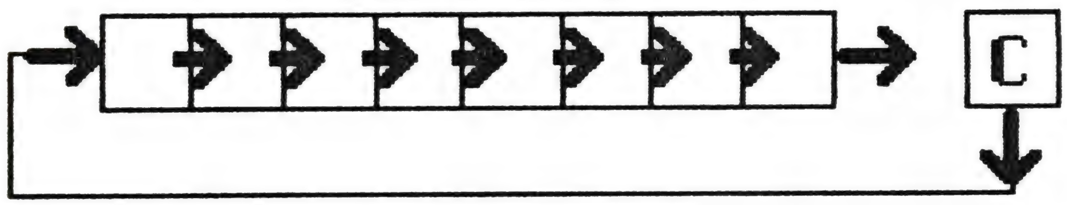
An arithmetic left shift is the same as a logical left shift, but an arithmetic right shift is different to a logical right shift because the vacant bit holds the value it had before the shift, rather than being filled with a zero.

Rotates work in the same way as logical shifts, except that the vacant bit is filled with the contents of the carry flag.

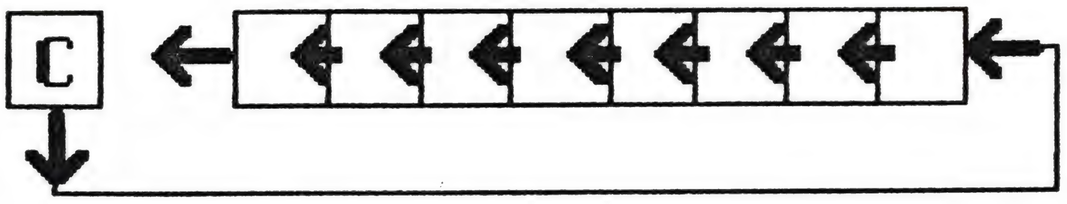
Arithmetic shift right



Rotate right



Rotate left



Shifts may be used for integer multiplication and division of *unsigned numbers*, i.e. numbers which are not in twos complement form and can only be positive.

A logical left shift is the same as multiplying by two, and a logical right shift is the same as dividing by 2.

Arithmetic shifts can be used in the same way to multiply and divide twos complement numbers, giving the quotient, with the exception being that dividing -1 by any value gives the result -1, not 0.

Copying an 8-bit number

As the 68000 can work with 8-bit, 16-bit, and 32-bit numbers, copying an 8-bit number into a 16-bit or 32-bit number (or 16-bit into 32-bit) can result in the loss of the twos complement properties of the number. To prevent this, the number is *sign-extended.* The most-significant bit of the number (the leftmost bit), also called the sign bit, is copied into all the remaining bits of the larger number.

e.g. If the 8-bit negative number 10001111 (-15) is copied into a 16-bit number, the result is 0000000010001111. The leftmost bit is now 0, signifying that the number is positive. Sign-extending the number gives 1111111110001111, a negative number.

Sign extension is automatically handled by the assembler when necessary.

Data formats accepted by the 68000

The 68000 instruction set can function with several different forms of data. These are binary, binary coded decimal (BCD), and ASCII.

Binary

Data in binary can be 8, 16, or 32 bits long. These three lengths are called *byte*, *word*, and *longword* respectively. Most binary data instructions can operate with any of the three lengths, with suffixes to indicate the length of the data.

e.g. The *compare* instruction **CMP** :

CMP.B	compares bytes of data
CMP.W	compares words of data
CMP.L	compares longwords of data

When a byte is stored in memory, it is stored in one location. However, when a word or longword is stored they take 2 and 4 locations respectively. The bytes are stored in decreasing magnitude, that is, the most significant bytes are stored first, at the lowest address, and the least significant bits are stored at the highest address.

e.g.#$2345 (hex) starting at $100 would be stored as two bytes, in locations $100 and $101. The contents of $100 would be 00100011, (23 hex) and those of $101 would be 01000101. (45 hex)

ASCII

This is probably thé most familiar form of coding used in computers today. ASCII - American Standard Code for Information Exchange - is used in all of the popular modern computers to represent characters. Each character is represented in memory and I/O operations by a numeric value which can be stored in one byte of memory. Up to 256 characters can be represented in this way, although the US standard ASCII defines only 128. Strings of characters are stored as a series of consecutive bytes, with an indication of the length of the string. Depending on the language being used, this length may be fixed, or variable. If it is variable, the end of the string may be denoted by a special character, or the length of the string may be stored at the beginning of the string.

BCD

Binary coded decimal is a data form which represents a decimal number as an equivalent four-bit binary number. It is similar to hexadecimal, without the digits A-F (10-15). It is frequently used for commercial applications, such as accounting, which require only addition and subtraction of numbers. Conversion from BCD to binary is simpler, and faster, than conversion from decimal to binary, and is thus far more efficient for simple manipulation of numbers. BCD is used in virtually all calculators and similar devices. A more detailed description of BCD is given in Chapter 13.

Floating point

Floating point format may be used to give more accuracy to numeric data, as well as increasing the range of numbers which may be dealt with by the computer. This type of representation is important for scientific applications, but is not directly supported by the 68000, in common with most other processors. It is, however, supported by many languages which have routines to produce floating point numbers. If you need to use floating point for an application, assembly language is not the language to use.

Writing assembly language programs

Included in this book are small assembly language programs which should work with the majority of assemblers. Assembly language programs are written in in much the same way as many high-level languages, as a sequence of instructions on numbered lines, which makes them very easy to correct and alter. There are, however, certain points which have to be observed when writing programs.

The first is that in all assembly programs, there has to be an instruction telling the assembler where to put the code that it is has translated. This instruction varies according to the assembler, and will be explained in the assembler manual. A little care is required when specifying the address where the code is to start, since the code must not be placed in certain areas in memory, such as the computer operating system or areas which contain data for the computer, otherwise some very strange results may occur! The space into which the program is assembled must be long enough for the code to fit, without extending into such areas as those previously mentioned.

Data for the program to operate on must also be stored in memory with care.

Free areas in memory should be specified in your assembler manual, unless the assembler has a special command to list free memory space. In place of any particular instruction, all complete programs in this book will have the word ORIGIN at the start of the program. This word should be replaced with the instruction relevant to your assembler, followed by the address where code is to start.

In order that the programs in this book should be easily adapted to any 68000 assembler and computer, certain routines which perform operations which vary according to the particular computer being used have been signified only by a label, such as routines to output text to the screen. These routines have not been defined in the programs, but they should placed in your programs, at the beginning.

These routines, once mastered, may also be placed in the other example programs in this book, to output answers to the screen, so that the effect of changing the operands in the programs may be more easily observed, without having to investigate the contents of memory where operands are stored.

Due to the diversity of computers and their different output systems, it is not possible to include print-routines etc. in this book for all 68000-based computers. The principles of creating these machine dependent routines are explained in Appendix 3.

Chapter Three
Arithmetic and Data

All arithmetic is performed in the ALU, as described in the previous chapter. Arithmetic operations may be performed on data directly given with the instruction, or it may be performed on data already stored in memory. Of the two, the latter is more usual. However, before the data in memory may be used, it has to be represented in the CPU, so that the ALU can access it. There are two ways in which the data can be made accessible to the ALU. For the moment, we will consider the direct way, which is to move it into a *data register*. Data registers are used as a temporary store for data to be used in an instruction. The 68000 has eight data registers, labelled **D0** to **D7**. These can be used to store byte (8-bit), word (16-bit) or longword (32-bit) quantities, but data shorter than 32-bits is not sign extended. When a byte or word length item of data is stored in a data register, only the 8 or 16-bits are affected, the rest of the register is unchanged.

e.g. If the register contents are #$EEEEEEEE, and the byte length data #$12 is moved into the register, the resulting contents are #$EEEEEE12.

To prevent any problems, it is best to *clear* (set all the bits to zero) a data register before moving an item of data into it, if the data is not a longword and is to be used for arithmetic. Clearing an address or register is done by the CLR instruction with an .L suffix, followed by the address of the location, or the label of the register, to be cleared.

e.g CLR.L D0 clears all 32 bits of data register 0. By placing .B or .W instead of .L, byte or word length sections may also be cleared.

Another method of clearing the register is to store the data as a longword, no matter what its actual length, since all bits not actually occupied by the number will be cleared.

The MOVE Instruction

Data is moved into a register with the MOVE instruction. Byte, word or longword data may be moved by placing .B, .W, or .L after the instruction as above. It is usually necessary to specify the length of data being moved in most instructions, since most can support various data lengths. If no data length is given, the computer assumes the data to be word length. This is true of most instructions, except certain special instructions which have no operands.

The syntax of the MOVE statement is:

MOVE.(data length) (source operand), (destination operand)

The source and destination are known as the *operands* of the instruction, the information about the data which has to follow the instruction in order that the operation can be carried out.

The source may be an address where data is stored in memory, it may be another data register, or it may be the data itself, in numeric form.

e.g MOVE.L $1005,D0 copies the longword data starting at address 1005 into data register 0. (Remember that a longword data item affects four bytes, or addresses). Note that the address is prefixed by a $ sign. This indicates that the address number is in hexadecimal notation. The contents of the address remain the same after the operation, since they are, in fact, copied, not actually moved.

MOVE.B D0,D1 copies the first 8 bits of the contents of data register 0 into the first eight bits of data register 1. D0 is unchanged by the operation, and the other three bytes of D1 are also unchanged.

MOVE.W #4660,D0 stores the word length decimal value 4660 in D0. The instruction MOVE.W #$1234,D0 does exactly the same, but the number to be stored is written in hexadecimal, as signified by the $ prefix. Examining the register after each operation would show exactly the same result, since 4660 (decimal) = 1234 (hexadecimal), and the contents of memory are displayed in hexadecimal.

The MOVE instruction can be used to MOVE virtually any data to any place in memory, not just the data registers. e.g MOVE.B #$12,$1000 moves the number $12 into address $1000.

Another version of the MOVE statement is the MOVEQ instruction, which moves a byte-length immediate data value (the number given in the instruction) into a data register. This instruction can only be used with a data register, it cannot be used to put data in any other part of memory.

Syntax : **MOVEQ #nn,Dn**

No data length needs to be specified. The number is sign-extended to fill all 32 bits of the register.

Since the number can only be one byte long, the range of numbers which can be used with this instruction is -128 to +127 (decimal).

Note: Most assemblers will convert a MOVE statement with the correct operands into a MOVEQ instruction during translation.

The Address Registers

The 68000 has another set of registers - the *address registers*. There are also eight of these, labelled **A0** to **A7**. The address registers may be used to hold 32-bit (longword) and 16-bit (word) quantities, but *cannot be used to hold 8-bit (byte)* *quantities*. When 16-bit quantities are stored in the address registers, they are sign-extended to 32-bits, to fill the register.

As the data registers are used to hold data for operations, so the address registers are used to store addresses of memory locations. The most significant byte of the address registers is ignored by the 68000, and so limits the memory which can be addressed by the 68000 to 2^{24} bytes - about 16 Mbytes (megabytes).

To put data into an address register, the MOVEA instruction is used. Only word and longword data may be specified with this instruction. Word data is sign-extended to 32-bits. The MOVEA has the same syntax as the MOVE intruction.

Unlike the data registers, which can only be used directly, the address registers can be used in a variety of ways, depending on whether it is the contents of a location or the address of the location which is required for the instruction.

These different ways of using the address registers are called *addressing modes*. They are used to inform the processor of the type of data on which the operation is to be performed, and where the data is held. This enables the processor to distinguish between two operations with the same mnemonic which use two different kinds of data. These are two examples using addressing modes met previously, although not identified as addressing modes.

e.g. MOVE.B #$23,D0 is an example of *immediate* addressing, and MOVE.B D1,D0 is an example of *data register direct* addressing. In the first example the data is immediately given in the instruction, and in the second example only the location of the data is given. The processor interprets these as two *different* instructions and assigns them different operation codes, although they have the same mnemonic. Without addressing modes, the instruction set would be considerably larger, with each instruction having a set of different mnemonics for each data source. Addressing modes can thus be thought of as an aid to humans, pandering to our reluctance to learn large numbers of instructions by grouping similar instructions together under the one mnemonic with variations in the operands to indicate the different data sources, leaving the assembler to recognise them as different operations.

The two basic addressing modes which apply to the address registers are called *address register direct* and *address register indirect*.

Address register direct is similar to data register direct, since it tells the processor that the address register itself is to be accessed.

e.g MOVE.L A0,D0 tells the processor that the data to be copied is held in the address register.

Address register indirect tells the processor that the location whose address is held in the address register is to accessed.

e.g MOVE.L (A0),D0 tells the processor to go to the address held in A0 and copy the contents of the address into D0. Thus the address register *indirectly* refers to the data.

The second method of making data accessible to the ALU can now be discussed. In this method, address register indirect addressing is used. The data to be used is stored in an address in memory, and the address is stored in an address register. The data is said to be 'pointed to' by the address register. When using this addressing mode, the address register label is always enclosed in brackets.

e.g Suppose A0 contains the address $1234.

ADD.B (A0),D0 tells the processor to add together the contents of address $1234 (the address 'pointed to' by A0) and the contents of D0.

This method is most used for data which has to be fetched from memory, changed, and then stored in the address from which it came from.

e.g. ADD.W D0,(A0) adds the contents of D0 to the contents of the address 'pointed to' by A0, and stores the result in that address.

All arithmetic operations in the 68000 operation set leave the result of the operation in the destination operand, although multiplication and division operations only allow the destination operand to be a data register.

Integer Arithmetic Operations

The 68000 instruction set contains a number of integer arithmetic operations which makes arithmetic somewhat simpler than with many other processors. The main operations are ADD, SUB, DIVS/U, and MULS/U. Arithmetic may be performed on binary and BCD values. This chapter deals only with binary arithmetic, for an explanation of BCD arithmetic see Chapter 13.

Before any addition or subtraction operations are performed, the carry flag should be cleared. The carry flag is a bit in a CPU register called the *condition codes register* (CCR). Each flag (bit) in this byte length register is set or cleared to indicate whether an event has occurred or not.

The different flags are :

N (negative flag) - indicates whether a twos complement number is positive or negative.

X (extend flag) - same as carry flag, but only affected by multiprecision operations - i.e. operations with two or more steps.

Z (zero flag) - indicates a zero result in an operation.

V (overflow flag) - indicates a result larger than the operand result.

The carry flag is set if an addition operation results in a 'carry' or a subtraction results in a 'borrow'. It is cleared if no carry or borrow has occurred. It may also be set when certain other instructions are executed. As addition or subtraction automatically includes any 'carry' or borrow indicated by the flag, the flag should be cleared before any operations are executed. Another flag which should be cleared is the extend flag, which behaves in a similar way to the carry flag. The flags are cleared by MOVEing a word value into the CCR. To clear the CCR completely, this value should be #0. Thus the instruction is MOVE.W #0,CCR. When moving a value to the CCR, that value should always be word length, although the upper byte is ignored.

Addition

Consider the statement ANS=2+2. This means add together two numbers, and place them in a third location. If ANS is location $1000, whose address is stored in address register A0, the assembly language program of this statement would be

```
10 ORIGIN
20 MOVE.B #2,D0          Put 2 in D0
30 MOVE.W #0,CCR         Clear carry flag
40 ADD.B #2,D0           Add 2 to the contents
                         of D0, store result in D0
50 MOVE.W #$1284,A0      Put address in address
                         register 0
60 MOVE.B D0, (A0)       Put result in ANS
70 END
```

The new instruction is the ADD instruction, which adds the source operand to the destination operand.

The main instruction for addition is ADD. This may be performed on byte, word or longword values, with the usual suffixes.

Data for addition may be represented in various ways (immediately, stored in a data register..etc), and there are several variations of the ADD instruction, each of which is used for different data forms.

The result of the addition is always stored in the destination operand of the instruction, and so *two numbers CANNOT be directly added together.* One of the numbers must be stored EITHER in a memory address OR in a register. *An instruction such as ADD.B #$12,#$24 will produce an error*, since the processor has nowhere to store the number.

ADD

This adds two numbers together, and stores the result in the destination operand. The source operand is unchanged by the operation. One of the operands *must* be a data register.

Syntax : **ADD.n Dn,ea**
ADD.n ea,Dn

'ea' implies an *effective address* which may be another *data register, an address register, a memory address, or numeric data* given with the instruction (see first example below).

Byte, word, and longword data sizes may be used.

e.g. ADD.W #$1234,D0 adds the immediate hex value 1234 to the contents of D0, and stores the result in D0. (immediate addressing)

ADD.L (A0),D1 adds the contents of the address 'pointed to' by A0 to the contents of D1, and stores the result in D1. (address register indirect addressing)

ADD.W A3,D5 adds the contents of A3 to the contents of D5, and stores the result in D5. (address register direct addressing)

Note : When the source is an address register, the data can only be a word or longword value.

ADD.B $1004,D2 adds the contents of address $1004 to the contents of D2, storing the result in D2. (Absolute addressing - the address where the data is stored is given in the instruction)

Note that the destination operand cannot be an address register.

ADDA

Add to Address register. This instruction adds an effective address to an address register.

Syntax : **ADDA.n ea,An**

Note that the address register must be the destination operand. Either word or longword data may be specified, but not byte length data. If byte length data is used, it must be specified as a word, sign-extended.

The result is stored in the address register, and is sign extended if it is a word value.

e.g. ADDA.W (A0),A1 adds the contents of the address 'pointed to' by A0 to the contents of A1, and the result is stored in A1.

The source operand can be any effective address.

ADDI

Add Immediate. The source operand is always numeric data. The destination can be a data register, or a memory address. Data may be byte, word, or longword values.

Syntax : **ADDI.n #nnnn,ea**

e.g. ADDI.B #$12,D0 adds the immediate data value 12 hex to the contents of D0, and stores the result in D0.

ADDI.W #4460,(A0) adds the immediate decimal value 4460 to the contents of the address 'pointed to' by A0.

ADDQ

Add Quick. This adds a 3-bit immediate value to an effective address which must be the destination operand. This instruction is similar to ADDI, but is executed faster since the data is only 3 bits long. Data may be given as a byte, word or longword quantity, but the data itself must be between 0 and 7 (inclusive). If the destination operand is an effective address, the data must be word or longword length.

Syntax : **ADDQ.n #n,ea**

e.g. ADDQ.B #3,D0 fast-adds the number 3 to the contents of D0. Only the first byte of D0 is affected.

ADDQ.W #4,A0 fast-adds 4 to the address stored in A0. The result is sign extended in A0.

ADDQ.L #7,(A0) fast-adds 7 to the contents of the address 'pointed to' by A0. The whole of the longword quantity starting at that address is affected by the operation.

Note that most assemblers will automatically convert an ADD instruction with appropriate operands to ADDI, ADDQ etc. as necessary. Thus the instruction ADD will usually suffice for all addition operations. However, an appreciation of the different versions of the instruction is valuable, since the machine code for each is different.

These addition instructions are all used with the assumption that both the numbers being added are 32 bits or less, and that the result is also 32 bits or less. This is not always the case though, especially when adding together two longwords, e.g. the entire contents of two data registers. Numbers larger than 32 bits have to be stored in 32 bit sections, dividing into longwords from the right, with any remaining high-order bits being stored as a byte, word or longword as necessary. The number is stored in memory with the high-order bits in the lowest address.

Consider the 72 bit number #$132639460461847138. To store this number starting at address $1000, the number is first divided into two low-order longwords and a high-order byte: 13 26394604 61847138. The number is then stored as a byte and two longwords, as follows :

```
$1000-$1001=#$13
$1002-$1005=#$26394604
$1006-$1009=#$61847138
```

To add together two numbers larger than 32 bits, or which will produce a number larger than 32 bits, the addition has to be done in at least two parts, using another addition instruction, ADDX, described below. The low-order part of the numbers is added first, and then the rest of the number is added, together with any carry from the addition of the previous section.

ADDX

ADD eXtended. By using this instruction with the ADD instruction, numbers with an addtion result greater than 32 bits can be added together. This is called *multiprecision addition.* The instruction can be used in two ways, either to add together two data registers, or to add together two memory locations. The instruction allows byte, word, and longword data. In order for the processor to correctly identify a zero result, the Z (zero) flag in the CCR should be set BEFORE the instructions are carried out. The Z flag is normally set if the result of an operation is zero, and cleared if the result is not zero. However, the ADDX instruction does NOT cause the Z flag to be set if the result is zero, although it does cause it to be cleared if the result is not zero. The reason for setting the Z flag before executing the ADDX command can be illustrated with this example :

Assume that the addition results in zero (e.g. adding positive and negative values). The zero flag is NOT changed by ADDX on a zero result. Thus, if the Z flag is initially clear, it will still be clear after the addition, and the processor assumes that the addition had a non-zero result. This may then lead it to give an incorrect result.

Syntax : **ADD.n Dn1,Dn2......ADDX.n Dn11,Dn21**
ADDX.n -(An1),-(An2)......ADDX.n -(An1),-(An2)

In a program, the ADDX (data register form) is preceded by an ADD statement to add the low-order bits of the numbers. *Do NOT clear the CCR between the ADD and ADDX,* since the second addition must include any carry from the first addition to produce a correct result. The memory location form requires two ADDX statements, one to add the low-order parts, and the second to add the high-order parts of the number. ADD cannot be used, since it requires one of the operands to be a data register. Again, do not clear the CCR between the two instructions. The memory location addition instruction introduces a new addressing mode, called *address register indirect with pre-decrement.* The address register contains an memory address, but *before* the data is copied from the address, the contents of the address register are decreased according to the data length specified in the instruction. A .B length will decrease the address in the register by 1, .W decreases it by 2, and .L decreases it by 4.

The address stored in the address register is the first address after the END of the data. e.g. For data stored in $3000-$3003 (4 bytes), the address stored is $3004.

Using this addressing mode means that the addresses for the data for the second part of the addition do not have to be placed in the address registers by the programmer. This addressing mode is particularly useful when using large numbers, larger than 32 bits which the computer cannot store as one data item, since it is left to the computer to calculate the addresses of the remaining bytes of the number.

For instance, when adding together two numbers of 64 bits, i.e. each number stored in two locations, the *low-order* addresses are added first with ADDX.L -(An1),-(An2), and then the *high-order* addresses are added, again with ADDX.L -(An1),-(An2).

Suppose the 64 bit hex number #$0012345678901234 is stored in addresses $100-$1007 (64 bits = 8 bytes), and is to be added to the 64 bit hex number #$0065432199876543 stored in $2000-$2007. The low-order bytes, #$78901234 and #$99876543, are added first. They are stored as longword values starting at $1004 and $2004. Thus they are added with ADDX.L -(A0),-(A1) where A0 holds $1008, and A1 holds $2008. The registers are decreased by 4 before the data is added, so that they point to the start of the low-order bytes. The result of the addition is stored starting at $2004, the address now stored in A1. The resulting carry is stored in the *extend flag*. Like the carry flag, the extend flag is a bit in the CCR which is set to the carry from the addition.

However, the extend flag is affected only by multiprecision instructions, unlike the carry flag which can be affected by most instructions. The high-order bytes, #$00123456 and #$00654321, are stored in $1000 and $2000 as longwords. These are added with the instruction ADDX.L -(A0),-(A1). The addresses in A0 and A1 are automatically *decreased* by another 4 bytes so that the registers point to $1000 and $2000, the start of the high-order bytes. The two numbers are added, together with the contents of the extend flag. The result of the addition is stored starting at $2000. The result of the whole addition is a 64 bit value which is stored in addresses $2000-$2007.

The assembly language program to perform the addition would be:

```
10   ORIGIN
20   MOVE.L#$78901234,$1004         Store low-order
                                    parts.
30   MOVE.L#$99876543,$2004
40   MOVE.B #123456,$1000           Store high-order
                                    parts.
50   MOVE.B #654321,$2000
60   MOVE.W #4,CCR                  Set Z flag,
                                    clear rest of
                                    CCR
70   MOVEA.L #$1008,A0              Put 1st address
                                    in A0
80   MOVEA.L #$2008,A1              Put 2nd address
                                    in A1
90   ADDX.L -(A0),-(A1)             Add low-order
                                    parts.
100   ADDX.L -(A0),-(A1)            Add high-order
                                    parts, with
                                    carry.

110   END
```

The locations involved would vary as follows :

<u>Initially</u>

Addresses:

$1000-$1003=00123456,
$2000-$2003=00654321,
$1004-$1007=78901234,
$2004-$2007=99876543,
extend flag=0

<u>After first addition</u>

$1000-$1003=00123456,
$2000-$2003=00654321,
$1004-$1007=78901234,
$2004-$2007=12177777,
extend flag=1

After second addition

$1000-$1003=00123456,
$2000-$2003=777778,
$1004-$1007=78901234,
$2004-$2007=12177777,
extend flag=0

When adding together two 64-bit numbers, each stored in two data registers, the registers containing the low-order bytes are added first with the ADD instruction, and the registers with the high-order bytes are added afterwards with the ADDX instruction.

e.g. #$00000000FFFFFFFF is stored in D2 and D3, and #$0000000011111111 is stored in D0 and D1. Note that these are both positive numbers, although the low-order registers contain negative numbers. D1 and D3 are added first with ADD.L D1,D3 and the result stored in D3, with any carry stored in the extend flag. D0 and D2 are then added together plus the *carry in the extend flag*. Although D0 and D2 are both zero, this second addition has to be done, to include the carry, otherwise the result would be negative. By including the extend carry in the second addition the final answer is positive, as expected when adding two positive numbers. The total result is stored in registers D2 and D3.

The assembly language program is similar to that for two memory locations:

```
10  ORIGIN
20  MOVE.L#$FFFFFFFF,D1      Store low-order
                             parts.
30  MOVE.L#$11111111,D3
40  MOVE.L #0,D0             Store high-order
                             parts.
50  MOVE.L #0,D2
60  MOVE.W #4,CCR            Set Z flag, clear
                             rest of CCR
70  ADD.L D1,D3              Add low-order parts
80  ADDX.L D0,D2             Add high order
                             parts, with any
                             carry in the extend
                             flag.
90 END
```

Thus the steps through the addition are as follows :

<u>Initially</u>

Register D0=00000000,
D1=FFFFFFFF,
D2=00000000,
D3=11111111,
extend flag=0

<u>After 1st addition</u>

D0=00000000,
D1=FFFFFFFF,
D2=00000000,
D3=11111110,
extend flag=1

<u>After 2nd addition</u>

D0=00000000,
D1=FFFFFFFF,
D2=00000001,
D3=11111110,
extend flag=0.

Note that although most assemblers will accept ADD instead of ADDI etc. where appropriate, ADD *cannot* be used instead of ADDX.

Subtraction

Consider the statement **3-2**. This subtracts 2 from 3. The assembly language program is:

```
10  ORIGIN
20  MOVE.B  #3,D0          Put 3 in D0
30  MOVE.W #0,CCR          Clear CCR
40  SUB.B   #2,D0          Subtract 2 from D0,
                           and leave result in D0

70  END
```

The new instruction is SUB, which subtracts two numbers, leaving the result in the destination operand.

The basic subtraction instruction is SUB, with variations similar to those in addition, i.e. SUBA, SUBI, SUBQ, SUBX. *The source operand is subtracted from the destination operand*, and as in addition, the result is stored in the destination operand. Thus, *two numbers CANNOT be directly subtracted. An instruction such as SUB.B #$1,#$23 will produce an error.*

The CCR should be cleared before any subtraction operations are executed, since subtraction includes any borrow in the carry flag or the extend flag.

SUB

This instruction performs subtraction with the contents of a data register. Thus one of the operands *must* be a data register, but it can be either source or destination. The other operand may be any effective address, except when the source operand is a data register. In this instance, the destination operand can be any effective address except an address register.

Data length can be byte, word or longword. If the source is an address register, only word and longword lengths can be used.

Syntax : **SUB.n Dn,ea**
SUB.n ea,Dn

e.g. SUB.B $1004,D0 subtracts the contents of address 1004 from the contents of D0, and stores the result in D0.

SUB.W #$1234,D1 subtracts the hex value 1234 from D1, leaving the result in D1.

SUBA

The SUBA instruction is used when the destination operand is an address register. The source operand is subtracted from the contents of the address register, and the result is left in the address register. The source operand can be any effective address.

Data must be given as word or longword length. If the data is a word, the result is sign-extended in the register.

Syntax : **SUBA.n ea,An**

e.g SUBA.L A1,A2 subtracts the contents of A1 from the contents of A2, leaving the result in A2.

SUBA.W #$1234,A0 subtracts the hex value 1234 from A0, leaving the result in A0.

SUBI

The immediate number given as the source operand is subtracted from the destination operand. The source operand is always an immediate value, and the destination can be any effective address except an address register.

Data can be byte,word or longword.

Syntax : **SUBI.n #nnnn,ea**

e.g. SUBI.B #$12, D0 subtracts $12 from the contents of D0, leaving the result in D0.

SUBI.L #$12,(A0) subtracts the hex value 12 from the contents of the address 'pointed to' by A0. The result is stored in that address, sign-extended to 32 bits if necessary.

SUBQ

This instruction is used to quickly subtract a small constant from an effective address. Data may be byte, word, or longword, unless the destination is an address register, when the data cannot be byte length.

Syntax : **SUBQ.n #n,ea**

The data must be 3 bits or less, i.e. between 0 and 7.

e.g. SUBQ.B #1, D0 fast-subtracts 1 from the contents of D0, leaving the result in D0.

SUBQ.B #4,(A0) fast-subtracts 4 from the contents of the address 'pointed to' by A0.

To subtract numbers larger than 32 bits, or to perform a subtraction which will have a result larger than 32 bits (e.g. subtracting two negative numbers), multiprecision subtraction is required. The instructions used are SUB, to form the low-order part of the instruction, and SUBX to form the high-order part. Do NOT clear the CCR between the SUB and SUBX instructions, since any 'borrow' from the first instruction will be held in the extend flag and must be included in the second subtraction to produce a correct result.

As with the ADD statement, most assemblers will accept SUB in place of SUBI, SUBQ etc. when used with the appropriate operands, but SUBX (described below) cannot be replaced by SUB.

SUBX

Similar to the equivalent addition instruction ADDX, this may be used either to subtract two data registers, or to subtract the contents of two memory locations. Data may be byte, word or longword. The result is placed in the destination operand. In order that the Z flag should correctly indicate whether or not the result of the operation is zero, it should be set before the instruction SUB is executed, since the Z flag is NOT set if the result is zero, although it is cleared if the result is not zero. (Refer back to ADDX for a further explanation, if necessary)

Syntax : **SUB.n Dn1,Dn2....SUBX.n Dn11,Dn21**
SUBX.n -(An1),-(An2).... SUBX.n -(An1),-(An2)

The data register form is preceded by a SUB operation to subtract the low-order parts of the numbers, and the address register form is executed with two ADDX instructions. As with ADDX, the memory location form uses address register indirect with pre-decrement addressing to access the data.

e.g. To subtract a 34-bit hex number 200000000 stored in addresses $1000-$1004 from the 34-bit hex number 3000FFFFF stored in addresses $2000-$2004.

The assembly language program would be:

```
10   ORIGIN
20   MOVE.L #0,$1001             Store low-order
                                 parts.
30   MOVE.L #$FFFFF,$2001
40   MOVE.B #2,$1000             Store high-order
                                 parts.
50   MOVE.B #3,$2000
60   MOVE.W #4,CCR               Set Z flag, clear
                                 rest of CCR
70   MOVEA.L #$1005,A0           Put 1st address in
                                 A0
80   MOVEA.L #$2005,A1           Put 2nd address in
                                 A1
90   SUBX.L  -(A0),-(A1)         Subt. low-order
                                 parts
100   SUBX.B  -(A0),-(A1)        Subt. high-order
                                 parts with any
                                 borrow in
                                 the extend flag.
110   END
```

Note that the CCR is NOT cleared between the subtractions. The result of the operation is stored in memory locations $2000-$2004.

To subtract the same two numbers, each stored in a data register, the assembly language program is similar to the above program, but is slightly shorter, since no addresses have to be stored to access the data.

```
10  ORIGIN
20  MOVE.L #0,D1           Store low-order parts.
30  MOVE.L #$FFFFF,D3
40  MOVE.B #2,D0           Store high-order
                           parts.
50  MOVE.B #3,D2
60  MOVE.W #4,CCR          Set Z flag, clear rest
                           of CCR
70  SUB.L  D1,D3           Subt. low-order parts.
80  SUBX.L D0,D2           Subt. high-order parts
                           with any borrow in the
                           extend flag

90  END
```

Multiplication

Multiplication can only be performed on a number stored in the low-order word of a data register. There are two multiplication instructions, MULS and MULU. No data length needs to be specified for either instruction, since they both assume 16-bit (word) data. The data register is always the destination operand, but the source operand can be any effective address except an address register. Both operations multiply the 16-bit value in the data register by the 16-bit value of the effective address, and store the 32-bit result in the data register.

MULS

Multiply Signed. This instruction multiplies the two operands using twos complement arithmetic. Thus the two operands are treated as signed integers, the most significant bit (leftmost bit) being used as a sign bit to distinguish postive and negative numbers. The range of numbers which can be multiplied with this operation is $7FFF to $8000 (hexadecimal), or 32767 to -32768 (decimal).

Syntax : **MULS ea,Dn**

The destination operand must always be a data register. If necessary, the result is sign extended to 32 bits.

For example, to multiply the 16 bit number $1234 by $5, the assembly language program would be:

```
10   ORIGIN
20   MOVE.W #$1234,D0          Put $1234 into D0
30   MULS   #$5,D0             Multiply by $5.
40   END
```

Before the multiplication, the data register contains $1234. Afterwards, the result of the multiplication, $5B04, is stored in the data register, sign-extended to 32 bits.

Other examples using the MULS instruction :

MULS (A0),D0 multiplies the low-order word contents of D0 by the contents of the address 'pointed to' by A0.

MULS $5678,D0 multiplies the low-order word contents of D0 by the contents of address $5678.

Note: the effective address cannot be an address register, since that contains a longword quantity.

MULU

This instruction also multiplies the low-order word of a data register by a word-length effective address, but assumes the data to be UNsigned, i.e. always positive. The most-significant bit of the data is treated as part of the number, NOT as a sign-bit. The range of numbers which can be multiplied by this instruction is $0000 to $FFFF (hexadecimal), or 0 to 65535 (decimal).

The MULU instruction is used in the same way as the MULS instruction.

Syntax :   **MULU ea,Dn**

As with MULS, the effective address *cannot* be an address register.

An example using MULU is constructed as follows:

```
10   ORIGIN
20   MOVE.W  #$8456,D0      Put $8456 in D0
30   MULU   #2,D0           Multiply by 2
40   END
```

Although $8456 would normally be treated as a negative hexadecimal word, this instruction uses the sign-bit as part of the number, and considers the whole number to be positive. Thus the result stored in D0 is $000108AC, and not $FFFF08AC as it would be if using the MULS instruction, which would treat $8456 as negative.

The MULU instruction is used when dealing with positive numbers larger than $7FFF, and the MULS instruction for all smaller numbers. The range of numbers which can be multiplied with these two numbers is 65535 to -32768.

Division

As with multiplication, division can be performed on signed or unsigned integers. The instructions used are DIVS and DIVU. The number to be divided is contained in a data register, and is always the destination operand. The source operand is the number by which the contents of the data register are to be divided. The number in the data register is 32 bits long, sign-extended if necessary. The dividing data must be 16 bits long, sign-extended if less. The result is given as two 16 bit integers, the quotient and the remainder. The quotient is stored as the low-order word of the data register, and the remainder is stored as the high-order word.

If an attempt is made to divide by zero, an error will occur, and program execution will be stopped. Chapter 12 gives more details of the process executed when an error occurs.

Be careful not to divide a very large number by a small number, for if the resulting quotient is longer than 16 bits, the V flag (overflow) in the CCR will be set, and the contents of the data register will remain unchanged.

Note : This event should be checked for in a program in which it may occur, since it can lead to unexpected results later in the program, due to the use of incorrectly processed data.

DIVS

Divide Signed integers. The range of longword numbers which can be divided is $7FFFFFFF to $80000000 (hex), or 2147483647 to -2147483648 (decimal). The range of word numbers which may be used for the divisor is $7FFF to $8000 (hex), 32767 to -32768 (decimal).

Syntax : **DIVS ea,Dn**

No data length needs to be specified since the processor assumes 32 bit data in the data register and 16 bit data as the divisor.

An example in assembly language using DIVS to divide two numbers:

```
10  ORIGIN
20  MOVE.L    #$100,D0    Put data in D0
30  DIVS    #$3,D0        Divide by 3
40  END
```

Note that the data to be divided must be copied into the data register as a 32 bit (longword) quantity.

Before the division, the contents of D0 are $100. After the division the contents are $00010055. $0001 is the remainder from the division, and $0055 is the quotient.

The effective address cannot be an address register, since that must contain a 32 bit quantity.

DIVU

Divide Unsigned integers. As in DIVS, the longword data to be divided is stored in a data register, and the word divisor is given by the effective address. No data length needs to be specified, as it is assumed.

Syntax : **DIVU ea,Dn**

The effective address cannot be an address register.

The range of numbers which can be divided is 4294967295 to 0 (decimal), or $FFFFFFFF to $0 (hex), since the sign-bit is considered part of the number.

As with DIVS, the quotient of the result of the division is stored as the low-order word of the data, and the remainder is stored as the high-order word.

To divide two unsigned integers :

```
10    ORIGIN
20    MOVE.L    #$456,D0     Put data in D0
30    DIVU    #$5,D0         Divide by 5
40    END
```

Note that the data to be divided must be copied into the data register as a 32 bit (longword) quantity, sign-extended if necessary.

Before the division, the contents of D0 are $456. After the division the contents are $000000DE. There is no remainder from the division, since $5 divides exactly into $456. The quotient, $00DE is stored as the low-order word of the data register.

If the quotient and remainder of a division, whether from DIVS or DIVU, are required as separate data, three more steps must be added to the program:

```
31    MOVE.W    D0,$1000  Puts quotient in $1000
32    SWAP D0             Swap two halves of D0
33    MOVE.W    D0,$1100  Put remainder in $1100
```

The new instruction SWAP exchanges the low-order word of a data register with the high-order word.

```
e.g.            D0 = $12345678
                SWAP D0
                D0= $56781234
```

SWAP does not need a data length to be specified, since it can only operate with words.

Thus the complete program would be:

```
10  ORIGIN
20  MOVE.L #$456,D0       Put data in D0
30  DIVU #$5,D0           Divide by 5
31  MOVE.W D0,$1000       Puts quotient in $1000
32  SWAP D0               Swap two halves of D0
33  MOVE.W D0,$1100       Put remainder in $1100
40  END
```

The program now divides the two numbers, stores the quotient in the low-order word of the data register as a word value starting at address $1000, swaps the high and low-order words in the data register to access the high-order word (the remainder) and stores the remainder as a word starting at address $1100.

SWAP can only be used to exchange the two words in a data register - it cannot be used to swap the two words of a longword stored in memory or in an address register. In order to swap the contents of either an address register or a stored longword, they must first be copied into a data register, and then the words may be swapped.

Chapter Four
Making Decisions

The feature which makes a computer stand out as more than a glorified typewriter is its ability to make decisions based on certain facts. At the heart of these decisions is the Condition Codes Register, the CCR briefly described in the previous chapter.

The CCR is the low-order byte of the *Status register*. The status register (SR) is a word-length register in the CPU. The high-order byte of the SR is called the *system byte*, and the eight bits are flags which hold information about the computer operating system. This byte is cannot normally be accessed by a programmer. However, the CCR can be read, or changed, by a program. *The five low-order bits of the CCR contain the X, N, Z, V, C flags.* The other three bits of the CCR are not used. The flags of the CCR are changed by the results of many instructions.

The Carry flag is set if an arithmetic operation results in a carry or a borrow. It is also altered by many other operations, such as comparisons between numbers.

The oVerflow flag is set if the result of an operation is too large to fit into the data space expected, or if the data given in an instruction is longer than the data quantity specified in the instruction.

The Zero flag is set when an operation has a zero result, and cleared if the result is not zero.

The Negative flag indicates whether a result was positive or negative. The most significant bit (high-order bit) of the result is copied into the negative flag. Thus, after a negative result the N flag will be set, and after a positive result it will be cleared.

The eXtend flag is set if an addition results in a carry, or a subtraction in a borrow. It is cleared if no carry or borrow occurs. This flag is used to indicate a borrow in multiprecision arithmetic because it is only affected by arithmetic operations, unlike the carry flag which can be altered by many instructions. Thus the two additions of a multiprecision operation do not have to follow one another, and the result of the low-order operation can be altered by the program, if necessary, before the high-order result is calculated.

By considering the contents of these flags, the processor can make decisions and affect the flow of a program. For instance, if the result of an operation is zero the processor stores one number in a particular location but, if the result is not zero, another number is stored in that location.

These decisions use an 'IF....THEN' structure, with the state of a specified flag as the condition.

e.g. Using a program model :

1 IF the carry flag is set, THEN go to instruction 4

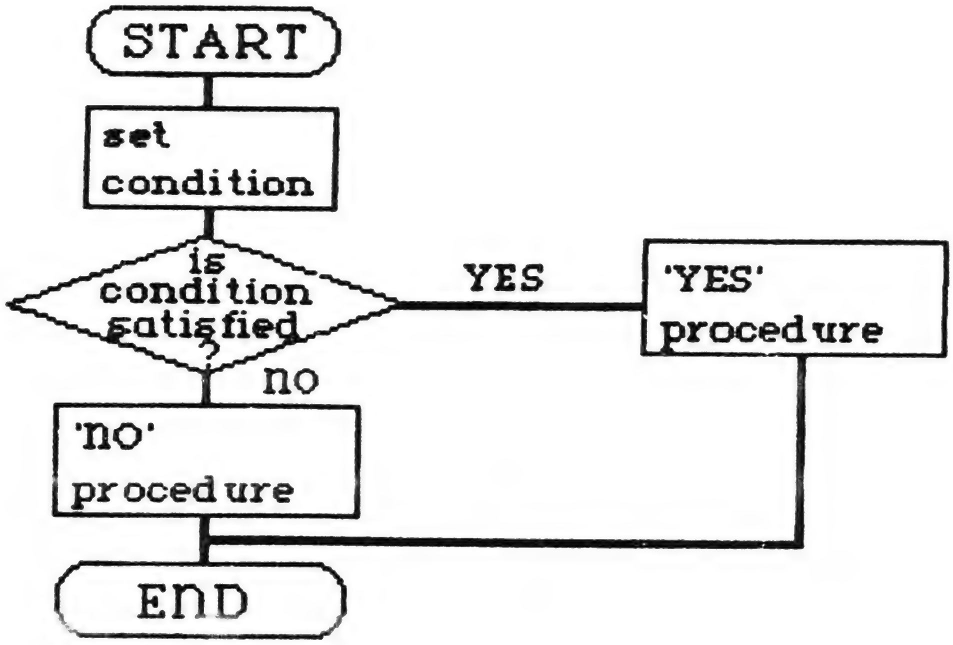
2 This instruction is executed ONLY if the carry flag is clear.

3 END.

4 The program branches to this location ONLY if the carry flag is set.

This model illustrates the way in which the instruction operates. The branch to another instruction is performed only if the condition is satisfied, and by branching, a number of instructions are missed. These instructions are only performed if the condition is not satisfied. Thus, these 'decision' instructions can be used to alter the outcome of a program, according to the state of a specified condition. It is this ability to modify the execution of a program that gives a processor its power and makes it so useful in its many various applications in today's technology.

The 'decision' instruction can also be represented by a flowchart diagram, as below.



These 'decision' instructions are also called *branching* instructions. There are 14 *conditional* branching instructions, and they are all of the form **Bcc.label.** The 'cc' is replaced in each instruction by an appropriate mnemonic, depending on the condition which is to form the basis of the branch. The 'label' is the address to which the program control is to branch if the condition is satisfied.

If the condition specified is satisfied, the displacement to the specified address is calculated and added to the program counter. The program counter is a 32 bit register in the CPU which always contains the address of the next instruction to be executed. When an instruction is executed, the PC is automatically incremented. It can otherwise only be affected by instructions such as branches which alter the order in which the program is executed.

Apart from testing the CCR flags, branches may also be performed according to the result of an arithmetic operation, or according to the result of a comparison between two registers, two memory locations, or between an immediate number and either an address or a register.

Comparisions are made between two quantities with the CMP instruction.

Syntax : **CMP.n ea,Dn**

Note that the destination operand must be a data register, although the source operand can be any effective address. Data size can be byte, word and longword.

When executing this instruction the processor subtracts the two numbers being compared, subtracting the first operand from the second, but does not alter the value of either of the operands. The result of the comparison is indicated by the state of the CCR. All the flags, except the eXtend flag, are affected. For instance, if the two numbers being compared were the same, the result of the subtraction would be zero, and the zero flag would be set.

Note that the SECOND operand is compared with the FIRST (i.e the first operand is subtracted from the second). Thus, if the second operand is greater than the first, the result would be positive and the Negative flag would be clear. If, on the other hand, the first number were greater than the second number, the subtraction result would be negative, and the Negative flag would be set. As a result of these changes in the flags, a conditional branch following a comparison can execute a branch if the SECOND operand was greater than, or if it was equal to, or if it was less than the first. Note that the result ALWAYS refers to the SECOND operand. These conditions may also be taken together, thus a branch may occur if the second operand is greater than OR equal to the first.

The comparison instructions may be used to test two quantities before an arithmetic operation is performed on them, in order that the operands may be in a particular order, e.g. to test which of two numbers is the larger, to ensure that the smaller is subtracted from the larger to avoid a negative result. After the comparison, if the first were larger than the second, the program would branch to an operation in which the operands were arranged to subtract the second from the first. Otherwise, if the second were larger, the program would continue without a branch, and subtract the first from the second.

This routine in assembly language would be :

```
10   ORIGIN
20   MOVE.L #$1000, A0       Put location in A0.
30   MOVE.L #$aaa,D1         Replace aaa by
                             number.
40   MOVE.L #$bbb,(A0)       Repeat for bbb.
50   CMP.L (A0),D1           Compare two
                             locations.
60   BGT.W branch.           location
                             If second bigger,
                             branch
70   SUB.L D1,(A0)           Do this if first is
                             bigger.
80   BRA  loc2               Go to end.
90   NOP                     branch.location
                             Branch to do this
                             only if
100   SUB.L (A0),D1          second is bigger.
110   NOP                    loc2
120   END
```

Replace 'aaa' and 'bbb' by any numbers to see the effect of the branches, by checking address $1000 and D1 to see which has been altered - remember that the result of a subtraction is stored in the destination operand.

Note that a label has been given instead of the actual location. This is because the actual location to which the program is to jump will vary, depending on the program origin. To find out the address of the location for the branch in your program, put any random number (also called a 'dummy' value) in line 30 in place of 'branch.location'. Now assemble the program to give a listing of both the program and the memory in which it is stored. Make a note of the location in which the NOP instruction is stored. (The machine code instruction for NOP is $4E71). Now replace the dummy value in line 30 with this address. The program will now run properly. This should be repeated for lines 50 and 80, to find the address of 'loc2' and ;branch.location, so that the program branches to the end (after performing the subtraction when the second operand is less than the first), bypassing the alternative subtraction. This method is the easiest way of calculating the address for a branch, and can be used for all branches. By using the NOP instruction, a 'marker' can be placed in the program to identify a particular location, without the processor executing an instruction or altering the CCR. When the processor encounters a NOP instruction, it simply continues with the next instruction - thus it is similar to a REM statement in BASIC, since the processor does nothing when it encounters the instruction.

Branching instructions

Syntax : **Bcc label**

All branching instructions have two data sizes, short and word (signified by .S or .W), depending on whether the displacement which is added to the PC is a short (byte) or word length quantity. However, it is not usually necessary to specify a data length, since the majority of assemblers will perform the selection automatically.

The displacements forwards or backwards which can be used to address memory locations are restricted to 32768 ($7FFF) backwards or 32766 ($8000) forwards, since the displacement can only be a maximum of 16 bits. Note that, since instructions always begin on a word boundary, the address for the branch must be an even number. A simple method of finding the address for the branch is given with the previous program.

All the following programs have labels, instead of data, which should be replaced by numbers. By repeating the program several times with different numbers each time, you can investigate the effect of the branches, checking the contents of both the operands in which the results may be stored. The manual with your assembler should tell you how to find the contents of an address or register.

BCC

Branch if Carry Clear. The carry flag is tested, and the branch is performed if the contents are zero. This instruction is particularly useful in arithmetic programs, in order to test whether a multiprecision operation needs to be performed.

e.g. When adding two longword quantities together whose values are unknown, a BCC operation after the first addition can be used to determine whether or not a second addition is needed to form the complete result. If the carry flag is clear, there is no carry from the operation and the addition is complete. Thus the program branches to the end address, since the condition has been satisfied.

However, if a carry has occurred, the condition will not be satisfied and no branch will occur. The program is continued uninterrupted to perform the second addition.

The assembly language program would be:

```
10  ORIGIN
20  MOVE.L    #$aaa,D1      Replace 'aaa' and
                            'bbb'
30  MOVE.L    #$bbb,D3      with your own no.s
40  CLR D0                  Clear D0......
50  CLR D2                  .....and D2.
60  ADD.L  D1,D3            Add two quantities.
70  BCC    end.address      If no carry, end.
80  ADDX.L  D0,D2           Add carry.
90  NOP                     end.address
100 END
```

The numbers being added are stored in D1 and D3, and D0 and D2 are clear. If the addition requires no carry (i.e. the result is $7FFFFFFF or less), it will be stored in D3, and the second addition will not be performed. If the result is larger, a carry is required, the second addition will be performed, and the high-order bits of the result will be stored in D2, with the low-order longword in D3. Check the contents of D2 after running the program to see what size numbers cause the branch to occur.

Note that the address to be substituted into line 30 is found as before, with the NOP instruction indicating the address for the branch. Remember to substitute 'end.address' with a dummy value for the first assembly.

BCS

Branch if Carry Set. This instruction performs the exact opposite to BCC, in that the branch only occurs if the carry flag is set. A branch will thus occur if a carry or borrow has occurred in an arithmetic operation. The carry flag will also be set after a CMP instruction, if the operand being tested by comparison is larger than the operand with which the comparison is made (i.e. a borrow has occurred in the comparison subtraction).

e.g. Two numbers are stored in data registers D0 and D1. The following program sorts them so that the smaller number is in D0, and the larger number is in D1. Replace 'aaa' and 'bbb' with different numbers to see the effect on the branch.

```
10    ORIGIN
20    MOVE.L    #$aaa,D1       Replace  'aaa'  and
                               'bbb'
30    MOVE.L    #$bbb,D0
40    CMP.L   D1,D0            Compare  numbers.
50    BCS     end.address      If  D1  larger  end.
60    MOVE.L  D1,D2            If  D0  is  larger,
70    MOVE.L   D0,D1           swap  the  two
80    MOVE.L   D2,D0           numbers.
90    NOP                      end.address
100   END
```

Find and substitute end.address as before.

The two numbers are compared. If they are in the right order, the program branches to the end. If they are not in the right order, the comparison subtraction will have resulted in a borrow, setting the carry flag. If the flag is set, the numbers are swapped.

BEQ

Branch if EQual to zero. If the Zero flag is set, this instruction causes the program to branch. The Z flag may be set if the two operands of a CMP instruction are equal, or if the result of an arithmetic operation is zero. The Z flag may also be set by the MOVE group of instructions, if the number being moved is zero. This last property is useful to detect if the divisor in a division is zero, before the division, thus avoiding an error in dividing by zero.

In the following program to test if the divisor is zero, both the divisor and the numerator are stored in memory, at locations $1000 and $1100.

```
10  ORIGIN
20  MOVE.L  #$aaa,$1000        Replace 'aaa' and
                               'bbb'
30  MOVE.L  #$bbb,$1100
40  MOVE.W  $1000, D0          Divisor.
50  BEQ  end.address           If zero, end.
60  MOVE.L  $1100,D1           Numerator
70  DIVS  D0,D1                Divide, if not
                               zero.
80  NOP                        end.address
90  END
```

Thus, if the divisor is zero, the program branches to the end, to avoid a error in dividing by zero. Remember to find and substitute 'end.address'. Replace 'aaa' and 'bbb' with several different numbers to see the effect on the branches.

BGE

Branch if Greater or Equal. This instruction causes a branch if the N and V flags are both clear, or both set. If the two flags are clear, then the result of the preceding comparison was positive, and was not larger than the two operands being compared - thus the operand to which the comparison was being made was larger, or equal to the other operand. If both flags are set, this indicates a large positive result, which overflows the data space allocated. The N flag is set because, in twos complement arithmetic, there is an extra bit at the left of the number which is set to 1 if a subtraction result is positive (see Chapter 2 if necessary). Although the subtraction instructions ignore the extra bit, the comparison instruction does not, since it is not a true subtraction instruction. The V flag is set because a subtraction in twos complement arithmetic always produces a result one bit longer than the longest operand.

This instruction generally follows a comparison instruction, and is used to distinguish numbers, performing one operation if the number is larger than OR equal to the number to which it is to be compared, and executing another operation if the number is less.

e.g. This program compares the contents of D0 to #$6, and ends if the contents are greater than or equal to #$6. If the number is less, #$6 is added to the number.

```
10    ORIGIN
20    MOVE.L   #$aaa,D0   Replace  'aaa'
30    CMP.L    #$6, D0    Is it  larger?
40    BGE   end.address   If  yes,  end.
50    ADD.L    #$6,DØ     If  not,  add  6.
60    NOP                 end.address
70    END
```

Find and replace end.address as before. Test several different numbers in D0, replacing 'aaa'. If the number is larger than 6, the program branches to the end, and the contents of D0 are unaffected. If the contents are less than 6, the program adds 6 to the contents of D0 before ending.

BGT

Branch if Greater Than. The branch occurs if the N, V, and Z flags are ALL clear, or if the Z flag is clear and the N and V flags are both set. This instruction acts in a similar way to BGE, but also checks the Z flag, not just the N and V flags. As before, the branch occurs if the N and V flags are both set or both clear. But, for this instruction to branch, the Z flag must also be clear. If the Z flag is set, the instruction will not branch. If the Z (zero) flag is clear, it indicates that the comparison subtraction result was not zero, and thus the numbers compared were not equal.

If the N and V flags are also both set or both clear, the second operand can be seen to be greater than the first. Thus the instruction will cause a branch if the comparison proves the second operand larger than the first, but will not branch if they are equal.

An example using BGT is shown as the first program in this chapter, on page 40.

BLE

Branch if Less than or Equal. This instruction executes a branch if the destination operand of a comparison is less than or equal to the source operand. Thus, the branch occurs if the Z bit is set, indicating a zero comparison subtraction result - i.e. the operands are equal. The branch also occurs if the N flag is set, and the V flag is clear, indicating that the comparison subtraction result was negative and thus that the destination operand is less than the source operand. The branch may also occur under a third set of conditions, if the N flag is clear and the V flag is set. This indicates that the comparison subtraction was negative and overflowed the allocated data space. Note that the leftmost bit of a twos complement subtraction is zero if the result is negative. The V flag is set because the answer is longer than the specified data length.

e.g. BLE can be used to check if a number is below a particular value. The following program checks to see if a number is between two limits, reducing or increasing as necessary.

```
 10  ORIGIN
 20  MOVE.L  #$aaa,D1         Replace 'aaa'
 30  CMP.B #$12,D1            Compare D1 to #$12.
 40  BLE  next.address        If second less,
                              branch.
 50  SUB.B #$12,D1            If larger, subtract.
 60  NOP                      next.address
 70  CMP.B #$6,D1             Compare to #$6.
 80  BGE  end.address         If larger, branch to
                              end.
 90  ADD.B #$6,D1             If less, add #$6
 100 NOP                      end.address
110  END
```

This program has two branches, to check if the byte-length number is between #$12 and #$6, inclusive. The first branch instruction checks to see if the number is below #$12. If not, #$12 is subtracted from it. The second branch instruction checks if the number is larger than #$6. If not, #$6 is added. Investigate the effect of different numbers in D1 on the branches, by replacing 'aaa' with several different numbers. Find next.address and end.address by replacing each with dummy values and assembling. Note the two addresses where the NOP statements are stored, and subtitute next.address by the first address, and end.address by the second.

BLT

Branch on Less Than. This is similar to BLE, except that no branch occurs if the operands are equal . The branch thus occurs if the N flag is set and the V flag is clear, or if the V flag is set and the N flag is clear - i.e. when the destination operand is less than the source operand.

e.g. The following program compares the contents of a data register to an immediate value, branching to the end if the contents are less than the value.

```
10  ORIGIN
20  MOVE.L    #$aaa,D0       Replace  'aaa'
30  CMP.L  #$3456F,D0       Compare  two  values.
40  BLT  end.address        Branch  if  less.
50  SUB.L  #$3456F,D0       Subtract  if  greater.
60  NOP                     end.address
70  END
```

Find end.address as before. The program compares the contents of DO to #$3456F, subtracting #$3456F if the contents are greater.

BHI

Branch if HIgher. This is used to test the result of a comparison between two UNSIGNED numbers. All numbers are treated as positive, and the branch occurs if the destination operand of the comparison has a higher value than the source operand. A negative subtraction result, indicating the destination is smaller, sets the carry flag since a borrow is used in the subtraction.

The branch occurs if the comparison subtraction was not zero and there was no borrow - i.e the two numbers were not equal, and the destination operand was larger than the source. Thus, the instruction branches if the Z flag and C flag are BOTH clear. The instruction is similar to the BGT instruction, but can be used for unsigned numbers since the N flag is not checked.

The range of postive numbers which can be compared is thus greatly increased by treating the most significant bit as part of the number, not as a sign bit. For instance, the byte value $9F is -97 (decimal) if treated as a signed byte value, since the most significant bit of the byte is 1. If considered unsigned, its value is 159 (decimal). One byte can represent a positive value up to 255 (decimal) if unsigned, compared with 127 (decimal) if signed.

e.g. To test which of two 32 bit unsigned numbers is larger, swapping them if the second is larger than the first. The first is stored in D0, the second in D1. Assume the registers already contain numbers.

```
10   ORIGIN
20   MOVE.L     #$aaa,D1      Replace  'aaa'  and
                              'bbb'
30   MOVE.L     #$bbb,D0
40   CMP  D1,D0               Compare  second  to
                              first
50   BHI    end.address       If  first  is  larger,
                              end.
60   MOVE.L  D0,D2            Swap  numbers  if  the
70   MOVE.L  D1,D0            second  is  larger.
80   MOVE.L  D2,D1
90   NOP                      end.address
100  END
```

Remember to replace 'end.address'. Note that although the CMP instruction assumes the numbers to be signed, they are treated as unsigned by the branch instruction, since it does not check the N and V flags.

BLS

Branch if Less or Same. This instruction is used with unsigned integers, in the same way that BLE is used with signed integers. The branch occurs if the C flag or the Z flag is set. Thus a branch occurs if a borrow has been used in the comparison subtraction, indicating that the destination operand is less than the source. A branch also occurs if the operands were the same, making the comparison subtraction result zero, and setting the Z flag. As with the BHI instruction, the range of positive integers which can be compared is doubled, since the sign bit is considered part of the number.

e.g. To test if the contents of D0 are less than or the same as the word number #$9456.

```
10   ORIGIN
20   MOVE.L     #$aaa,D0     Replace  'aaa'
30   CMP.W   #$9456,D0       Compare  two  values.
40   BLS    end.address      Branch  if  less  or
                             equal.
50   MOVE.W   #$9456,D0      If  larger,  set  to
                             #$9456
60   NOP                     end.address
70   END
```

The number #$9456 is treated as an unsigned word integer, 37974 (decimal). If the contents of D0 are less than or equal to #$9456, they are unchanged. If the contents are larger, they are replaced by #$9456. Find 'end.address' as before.

BMI

Branch if MInus. If the N flag is set, the branch is executed. This instruction can be used to test if the result of the previous expession was negative, and branch if the condition was fulfilled.

This branch instruction should not be used after a comparison to test if the destination operand was smaller than the source, although the N flag would be set if the comparison subtraction was negative, indicating a smaller destination operand. Examining the conditions for the instructions BGE and BGT will demonstrate that the N flag can be set by a positive result of a comparison subtraction. To branch on a smaller destination operand, the BLT instruction should always be used, since it also tests the V flag to confirm a negative subtraction result.

The BMI instruction can be used to test if a number is negative after an instruction such as MOVE, SUB or ADD, all of which set the N flag if the result of the operation was negative.

e.g. A number is stored as a byte at address $1000. The number is moved into a data register. This program tests whether or not the number is negative. If the number is positive, it has #$5 subtracted from it.

```
10  ORIGIN
20  MOVE.B  #$aaa,$1000      Replace 'aaa'
30  MOVE.B  $1000,D0         Move into D0.
40  BMI end.address          If negative,
                             branch.
50  SUB.B  #$5,D0            If positive,
                             subtract.
60  NOP                      end.address
70  END
```

Find 'end.address' as before. After the value has been copied into D0, in line 20, if it is negative then the N flag is set, otherwise the flag is cleared. The BMI instruction tests the N flag, and branches if it is set.

BNE

Branch if Not Equal. The Z flag is tested. If it is clear, the branch is executed. The Z flag is set if the result of the previous operation was zero, and is cleared if the result was not zero. Thus, this instruction executes a branch if the result of the previous operation was not zero. This instruction is the opposite of the BEQ instruction, and usually follows a CMP instruction, to perform a branch if the two operands are not the same.

e.g. The following program compares two operands. If they are different, the program branches to the end. if they are the same, #$10 is added to the second operand.

```
10    ORIGIN
20    MOVE.L    #$aaa,D1     Replace 'aaa' and
                             'bbb'
30    MOVE.L    #$bbb,D0
40    CMP.L   D1,D0          Compare D1 to D0.
50    BNE  end.address       Branch if not equal.
60    ADD.L #$10,D0          If equal, add #$10 to
                             second operand.
70    NOP                    end.address
80    END
```

Find 'end.address' as before.

BPL

Branch if PLus. The branch is executed if the result of the previous operation was greater than zero - i.e. If the N flag is clear.

This instruction may be used after such instructions as MOVE,ADD and SUB. It should not be used after CMP to test that the subtraction result was positive, indicating that the destination operand was larger. The explanation of the conditions of BGT show that the N flag may be set after a positive comparison subtraction. BGT should always be used after a comparison which checks for a greater value, since BGT confirms a positive result with the contents of the V flag.

e.g. In this program, the BPL instruction is used to test whether a number is positive. If it is, unsigned multiplication is used to multiply by #$5. If it is not positive, it is multiplied by #$5, using signed multiplication.

```
10    ORIGIN
20    MOVE.W  aaa,  D0        Replace  'aaa'
30    BPL  u.mul              Branch  if  positive.
40    MULS  #$5,D0            Multiply  signed.
50    BRA  end.address        Branch  to  end.
60    NOP                     'u.mul'  address
70    MULU    #$5,D0          Multiply  unsigned.
80    NOP                     'end.address'
90    END
```

The addresses 'u.mul' and 'end.address' are found as described before.

BRA

A new branch instruction, BRA, is introduced in line 50 of the above program. This is an *unconditional branch*, and is used to branch from one part of a program to miss out a section in the same way as all the branches so far discussed, in this case to avoid the multiplication which only takes place if the operand is positive. Unlike the branches so far investigated, BRA has no conditions and the branch is *always* executed. Thus if two routines occur consecutively in a program, each to be executed on two different types of data - e.g. the first routine is for positive data, and the second for negative data - the first routine is branched over by the conditional branch if the data is positive.

If the data is negative, the first routine is executed. However, when the end of the first routine is reached the program must branch over the second routine to reach the end, in order to avoid executing the second routine as well as the first. In this case, since the branch must always be made, the instruction for the branch is BRA. An 'END' statement *cannot* be placed in the program at this point, since END is not a 68000 instruction, but merely an indicator to the assembler to finish assembling the program. If an END statement were placed in the program at this point, the assembler would take it as the end of the program, and would not assemble the rest of the program (lines 50 - 90) into machine code. Thus a BRA instruction must be used.

BVC

Branch if V flag (overflow) clear. The instruction branches if the previous operation result did not overflow the data space specified for it, whether it was a byte, word, or longword.

e.g. This program tests to see if the result of an addition can be stored as a byte. If not, it is stored as a word.

```
10  ORIGIN
20  MOVE.W aaa,D0             Replace 'aaa' and
                              'bbb'
30  MOVE.W bbb,$1000
40  MOVE.W #$1000,A0
50  ADD.B (A0),D0             Add values
60  BVC end.address           Branch to end if
                              fits.
70  MOVE.W aaa,D0             Reset D0.
80  ADD.W (A0), D0            Add values again, in
                              a larger data space.
90  NOP                       end.address
100 END
```

Find 'end.address' as before.

BVS

Branch if V flag set. This instruction is the opposite to BVC, since it executes a branch if an overflow has occurred.

e.g. If an addition overflows, the two numbers are subtracted instead of being added again.

```
10   ORIGIN
20   MOVE.W  aaa,D0           Replace  'aaa'  and
                              'bbb'
30   MOVE.W  bbb,$1000
40   MOVE.W  #$1000,A0
50   ADD.B  (A0),D0           Add  values
60   BVS  sub.address         Branch  to  subtract  if
                              overflow.
70   MOVE.W  aaa,D0           Reset  D0.
80   ADD.B  (A0),  D0         Add  values  again.
90   BRA  end.address         Branch  to  end.
100  NOP                      'sub.address'
110  SUB.B    (A0),D0         Subtract.
120  NOP                      'end.address'
130  END
```

Find 'sub.address' and 'end.address' as before. Note that BRA is used again to branch over the second routine.

One unconditional branch has already been discussed, BRA. There is another, similar instruction.

BSR

Branch to SubRoutine. Like BRA, no conditions are required, the branch is ALWAYS executed. However, with this instruction, the contents of the PC are stored in memory by the processor before the displacement is added. The program then branches to a *subroutine*, a program section which executes a series of instructions. At the end of the subroutine there is a **RTS** (Return from SubRoutine) instruction. On encoutering this instruction, the processor restores the contents of the PC from memory, and the program continues to be executed from the instruction following the subroutine branch. Subroutines are used when a particular set of instructions are needed several times in one program. By writing the set as a subroutine, it can be branched to each time it is needed, eliminating the need to write it several times in the program, and also saving memory space.

e.g. The following program adds #$5 to the low-order byte contents of three memory locations. The addition is written as a subroutine, to be branched to when necessary.

```
10  ORIGIN
20  MOVE.B #$12,$1000
30  MOVE.B #$24,$1001
40  MOVE.B #$35,$1002
50  MOVE.B $1000,D0              Put contents
                                 into D0
60  BSR subroutine.address       Branch to add.
70  MOVE.B $1001,D0              Put contents
                                 into D0
80  BSR subroutine.address       Branch to add.
90  MOVE.B $1002,D0              Put contents
                                 into D0
100  BSR subroutine.address      Branch to add.
110  BRA end.address             Branch over sub.
                                 to end
120  NOP                         'subroutine.
                                          address'
```

```
130   ADD.B #$5,D0              Add #$5 to
                                contents.
140   RTS                       Return
150   NOP                       'end.address'
160   END
```

Lines 120-130 are the subroutine, and it is these lines to which the program branches each time the BSR instruction is executed. Find 'subroutine.address' and 'end.address' as before, and remember to substitute 'subroutine.address' in all three BSR instructions.

Jumps

There is another group of instructions which is very similar to the unconditional branches, the jumps.

Jumps execute in the same way as unconditional branches, that is, they jump to the address specified in the instruction without any condition having to be fulfilled. Unlike the branches, *jumps are not limited to a particular range of addresses in either direction.* Jumps can take place, both backwards and forwards, through any size of displacement (up to 32 bits, since that is the maximum displacment which can be stored - it is also the largest possible difference between two memory locations referenced by the 68000). Also, unlike branches, the address to which the jump takes place does not have to be directly given in the instruction, it can be stored in a memory address or a data register.

There are two jump instructions.

JMP

JuMP. The JMP instruction transfers program control to an effective address. The address may be stored in a data register or a memory location. The memory location can be accessed directly, as in a BRA instruction, or it can be 'pointed to', using address register indirect addressing.

Syntax : **JMP ea**

No data length needs to be specified. Since the effective address can be stored in a longword location, it can be up to 32 bits long. Note that the 68000 address bus ignores the high-order byte of the longword, so in practice only addresses up to 24 bits long can be accessed either by the PC or by a JMP instruction. Thus the JMP instruction can execute a jump to any location that the PC can access - that is, any address in the computer memory. The JMP instruction is used in the same way as the BRA instruction, with the differences previously mentioned in the way the address is stored, to jump to an address outside the range of the branch instructions, or to an address which cannot be found in the same way as that described before. e.g. If the jump can be to one of various addresses, depending to particular conditions, the address can be different each time the program is run, and it cannot be found from an assembly. In this instance, the address for the jump has to be stored in an address register after calculation, and the JMP instruction address is 'pointed to', by an address register. e.g. JMP (A0) jumps to the location whose address is stored in address register A0.

Note: The address for the jump must be a memory address. e.g. JMP D5 is an illegal instruction, since the program cannot jump to a data register. JMP A5 would also be illegal, since it refers to an address register. JMP (A5) is legal though, since it is a memory address which is being referenced, not an address register.

An example using the JMP instruction follows. This program uses JMP instead of BRA, and the memory address for the jump is stored in an address register, A0.

```
10   ORIGIN
20   MOVE.B #$12,$1000
30   MOVE.B #$24,$1001
40   MOVE.B #$35,$1002
50   MOVE.B $1000,D0               Put contents
                                   into D0
60   BSR subroutine.address        Branch to add.
70   MOVE.B $1001,D0               Put contents
                                   into D0
80   BSR subroutine.address        Branch to add.
90   MOVE.B $1002,D0               Put contents
                                   into D0
100   BSR subroutine.address       Branch to add.
110   MOVEA.L end.address,A0       Store address in
                                   A0.
120   JMP (A0)                     Jump over sub.
                                   to end
130   NOP                          'subroutine.
                                        address'
140   ADD.B #$5,D0                 Add #$5 to
                                   contents.
150   RTS                          Return
160   NOP                          'end.address'
170   END
```

Note: Find 'end.address' as before.

The JMP instruction is very useful in overcoming the limitations on the size of the displacement in the Bcc instructions. Since the branch displacement has a maximum size of a word, the displacement can only be 32768 bytes backwards, or 32766 bytes forwards. For a conditional branch of a larger displacement than this, the branch instruction passes to a JMP instruction which then jumps to the address to which the program is to pass from the branch. The JMP instruction is used as a 'stepping stone' to the address.

The following sequence shows how the JMP instruction is incorporated and called.

```
:                              Program
JMP  bcc.instr.

Jump  over  2nd  JMP.
Stepping.Stone
JMP  end.address
NOP                            bcc.instr  address
Bcc  Stepping  Stone
Branch  back  to  label
:
:
NOP                            end.address
END
```

The 'Stepping.Stone' JMP is usually defined *before* the Bcc instruction uses it. This means that the JMP can be called with a label, without having to find the address. Since the label 'Stepping.Stone' has been defined in the program before it is used, the processor has already marked an address as representing the label. The JMP bcc.instr is necessary to avoid executing the 'Stepping.Stone' instructions before the Bcc instruction has been executed. Thus, the program jumps over the 'Stepping. Stone' to the Bcc instruction, and then jumps back if the branch is to be executed.

JSR

Jump to SubRoutine. This instruction is similar to the BSR instruction. As with the JMP instruction, this instruction can jump to any address in computer memory, with the address to be jumped to either given directly, or stored in a data register or memory address, in the same way as a JMP instruction.

Syntax : **JSR ea**

No data length needs to be given.

An example using both JMP and JSR instructions follows. This is the program given above, but with the BSR instructions replaced with JSR. The address to be jumped to is held in address register A0.

```
10   ORIGIN
20   MOVE.B #$12,$1000
30   MOVE.B #$24,$1001
40   MOVE.B #$35,$1002
50   MOVE.L subroutine.address,A0
                                  Put sub.rt add.
                                  in A0
60   MOVE.B $1000,D0              Put contents
                                  into D0
70   JSR (A0)                     Jump to
                                  subroutine.
80   MOVE.B $1001,D0              Put contents
                                  into D0
90   JSR (A0)                     Jump to
                                  subroutine.
100  MOVE.B $1002,D0              Put contents
                                  into D0
110  JSR (A0)                     Jump to
                                  subroutine.
120  MOVE.L end.address,D5        Store address in
                                  D0.
130  JMP D5                       Jump over sub.
                                  to end
140  NOP                          'subroutine.
                                      address'
150  ADD.B #$5,D0                 Add #$5 to
                                  contents.
160  RTS                          Return
170  NOP                          'end.address'
180  END
```

Note: Find 'subroutine.address' and 'end.address' as before.

Summary of branch status of flags:

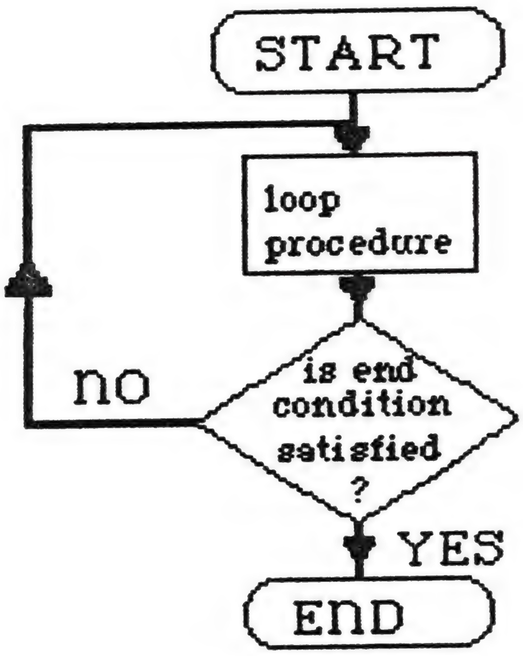
INSTRUCTION FLAG STATUS
1= SET, 0=CLEAR, .=UNTESTED

		X	N	V	Z	C
BCC		.	.	.	.	0
BCS		.	.	.	.	1
BEQ		.	.	.	1	.
BNE		.	.	.	0	.
BGE		.	1	1	.	.
	OR	.	0	0	.	.
BGT		.	1	1	0	.
	OR	.	0	0	0	.
BLE		.	1	0	.	.
	OR	.	0	1	.	.
	OR	.	.	.	1	.
BLT		.	1	0	.	.
	OR	.	0	1	.	.
BHI		.	.	.	0	0
BLS		.	.	.	.	1
	OR	.	.	.	1	.
BPL		.	0	.	.	.
BVS		.	.	1	.	.
BVC		.	.	0	.	.

Chapter Five
Loops and Looping

The branching instructions discussed in the previous chapter can be used in a program to perform one other task which was not mentioned, looping.

Loops are sections of programs which are repeated several times, until a termination condition is fulfilled.



This termination condition may be a count - e.g. loop seven times, then continue with the rest of the program. It may also be a condition such as those described for the branches - e.g. loop until the Z flag is set, then continue with the program. The loop may also be one with no end, which loops round the section for ever. This type of loop is not very useful, and is best to be avoided!

The branch instructions may be used for loops in the following way :

```
10   ORIGIN
20   MOVE.B    aaa,  D0      Replace   'aaa'.
30   Subtract.LOOP           Define  label.
40   SUB.B  #$1,  D0         Subtract  1  from  D0.
50   BNE   Subtract.LOOP     If  not  zero,  go  back.
60   END
```

By replacing 'aaa' with different numbers, you can test the effect on the loop, which is lines 30-50. The program subtracts #$1 from the low-order byte contents of D0 until the contents are zero. The label 'Subtract.LOOP' does not need to be replaced with an address, since the assembler has encountered the LOOP label definition in line 30, and has defined it as that address. Thus the processor already knows the address to jump to. Placing a label at the start of a line usually indicates to the assembler that the label is to be defined as the current address in the program counter. However, the label has to be defined BEFORE it can be branched to, thus labels can only be used for branches BACKWARDS, not forwards, unless the label is defined using another method of definition, by using a particular definition instruction, such as EQU. The syntax is 'label' EQU address. This instruction will vary according to the assembler being used.

However, the address for the instruction still has to be found in the same way as previously described. The main advantage to using labels is that it makes the program easier to read and understand, since an assembly listing may not show the addresses in which the program is stored, making it difficult to tell where a branch jumps to, if the destination is given as a number rather than as an address. In the case of loops and backward jumps and branches, the label is defined before use, so it is not necessary to define the label at the start of the program. The addresses of the labels for forward jumps are found as previously described - using an NOP instruction as the marker. Some assemblers do not allow the use of labels, but these are few and far between, and are best avoided.

Branch instructions should only be used for loops if the condition is certain to be satisfied. If the condition is never satisfied, the program will loop around the section indefinitely - an infinite loop that never ends.

There is another set of instructions, similar to the branch instructions, which is used for loops. These are the DBcc instructions - like the branch instructions, the 'cc' is replaced by a condition, but unlike branches, the condition is for the loop to end. DBcc instructions have one more operand than the branch instructions.

Syntax : **DBcc Dn, label**

The data register operand initially contains a value, which is the maximum number of times the loop is to be executed if the condition is satisfied during all of the loops. For instance, if the loop is to end either when a non-zero result occurs, or when it has looped 10 times, the data register will contain the value 10 at the beginning of the loop. The value of the contents of the data register will be decremented each time the DBNE instruction is executed. If the condition is satisfied, the loop will be executed, decreasing the contents each time, until the value of the contents of the data register are -1. Thus, when the processor encounters a DBcc instruction, it checks first to see if the condition to end the loop has been satisfied. If it has not, it then compares the contents of the data register to -1. If the contents are not equal to -1, the contents are decremented by 1, and the loop is executed again. If either the condition is satisfied, or the contents of the data register are equal to -1, the loop is ended. Thus the DBcc instruction enables a loop to be executed with an termination condition which may never be satisfied, but avoids infinite loops.

If the contents of the data register are initially equal to -1, the program will loop around the section 65536 times before ending.

Labels are usually used to signify the start of a loop, since they can then be used as the branch address without having to calculate the address of the start of the loop. It is best to make these labels as meaningful to the loop as possible. e.g. Instead of just calling a loop 'LOOP1', make the label relevant to the loop, such as 'SUBTRACT.LOOP', which indicates the purpose of the loop. This makes it much easier to read and understand the program.

DBcc - Decrement, and then Branch unless termination condition satisfied.

The condition specified in the instruction is that on which the termination is to occur.

e.g. To execute the loop again if a non-zero result occurs, the loop must end if the result is zero. Thus the instruction would be DBEQ - loop unless the result is zero. The loop ends when the condition is satisfied - i.e. when the result IS zero. All DBcc instructions assume word length data. Thus the range of addresses which can be branched to is the same as that for branches - no more than 32768 bytes backwards from the address at which the DBcc instruction is stored. DBcc instructions are not used to jump forwards, since the loop structure requires the termination condition for the loop to be at the end of the loop. Thus, the DBcc instruction must always be the last instruction in the loop section.

Note : When a condition is satisfied, it is said to be 'TRUE'. If it is NOT satisfied, it is said to be 'FALSE'. Thus, if the condition is TRUE, the loop ends, but if it is FALSE, the loop is executed again.

There are 17 DBcc instructions - 16 conditional, and one unconditional.

DBRA

Decrement and Branch Always - until the loop count is ended. This is an unconditional loop (no specified termination condition), so the loop is executed until the contents of the data register are -1.

e.g. This program stores the value #$12 in consecutive bytes in memory 10 times.

```
10  ORIGIN
20  MOVE.W #$1000,A0          Starting address.
30  MOVE.B #$9,D0             Store loop count.
40  Store.LOOP                Loop start.
50  MOVE.B #$12, (A0)+        Store and
                              increment.
60  DBRA D0, Store.LOOP       Branch if loop
                              count not -1.
70  END
```

Note that, since the loop is done until D0 is -1, the loop count stored in D0 is one less than the number of times the loop is to be executed.

Line 50 introduces a new addressing mode, *address register indirect with post-increment.* This is similar to the addressing mode discussed in the ADDX and SUBX instructions - address register indirect with pre-decrement. This new addressing mode performs a similar function, enabling a programmer to access a series of consecutive locations without needing to store the address of each location. The difference between the two is that, whereas a.r.i.w.pre-decrement counted down a series of addresses, from highest to lowest, a.r.i.w.post-increment counts up a series, from lowest to highest. *When using the (An)+ operand, the processor performs the operation on the address 'pointed to' by the address register, and then increments the address register* by 1, 2, or 4, depending on whether the data of the instruction was a byte, a word, or a longword, respectively.

The first address to be accessed is stored in the address register. Thus, in the loop section in the above program (lines 40-60), the first address to have #$12 stored in it is $1000. The processor then increases the contents of A0 by 1, since the operands were byte size. Thus, the next time the loop is executed, the contents of A0 are $1001, and #$12 is then stored in $1001. The contents are again increased by 1, and the process continues each time the loop is executed, to the end of the program.

Thus, address register indirect with post-increment can be thought of as, execute the task on the contents of the address in the address register, then increase the contents of the address register .

The conditional loop branches take the same end mnemonics as the conditional branch instructions.

DBCC

Decrement and Branch until the Carry is Clear. Used for a loop which is to end when the carry flag is clear. Thus, if the carry flag is set, the loop is executed again.

e.g. The following program subtracts two numbers. The first is decremented at the start of the loop. If the first (the source) is larger than the second, a borrow occurs in the subtraction and the condition to repeat the loop is satisfied - i.e. the carry flag is set. The loop is repeated until the source is the same as the destination, and no borrow occurs, clearing the carry flag.

```
10   ORIGIN
20   MOVE.B aaa, D0                 Replace 'aaa' &
                                    'bbb'.
30   MOVE.B bbb, D1
40   MOVE.B #$F,D2                  16 loop limit
50   Subtract.LOOP                  Loop start.
60   SUBQ #$1, D0                   Decrement D0.
70   SUB.B D0, D1                   Subtract.
80   DBCC D2, Subtract.LOOP         Branch if borrow.
90   END
```

By placing different values in D0 and D1, the effect on the loop can be investigated. If D0 is more than D1+16, the loop will terminate before the condition is satisfied.

DBCS

Decrement and Branch until Carry Set. This is used in a loop which is to end when the carry flag should be set. Thus, the DBCS instruction ensures that the loop is executed while the carry flag is clear, and ends either when the carry flag is set, or when the loop count in the data register has reached -1.

e.g. The following program compares the byte contents of two addresses. If the contents of the first are larger than the second, the comparison subtraction results in a borrow, setting the carry flag, ending the loop. The contents of the two addresses are swapped, and the loop is branched into again. The loop finally ends after 64 pairs of addresses have been compared.

```
10   ORIGIN
20   MOVEA.W #$1000,A0         Start address #2.
30   MOVEA.W #$2000,A1         Start address #2.
40   MOVE #$40,D0              Loop counter.
50   Compare.LOOP              Loop start
60   Jump.point                Address for jump
                               back.
70   CMPM.B (A0)+,(A1)+        Compare memory.
80   DBCS D0,Compare.LOOP      End loop if carry
                               set.
90   CMPI.B #$FF,D0            Is count at -1?
100  BEQ end.address           If yes, loop count
                               has ended, so goto
                               end.
110  MOVE.B -(A0),D0           If not, carry is
                               set, so swap data.
120  MOVE.B  -(A1),(A0)
130  MOVE.B  D0, (A1)
140  MOVE.B  #0, CCR           Clear CCR.
150  JMP Jump.point            Jump into loop.
160  NOP                       end.address
170  END
```

Substitute 'end.address' with the relevant NOP address, as explained previously. The other labels in the program - 'Jump.point' and 'Compare.LOOP' - have been defined before they are used in an instruction, so the processor already has an address reference for them. The CMPI and BEQ instructions in line 90 and 100 are to ensure that, when the loop count is ended, the program does not exchange the last two address contents and jump back into the program. Note that #$FF (hex) = -1 (decimal) in twos complement arithmetic.

Two new forms of the CMP instruction are introduced in this program, CMP and CMPI. CMPM is used to compare the contents of two memory addresses, and the addresses must ALWAYS be referred to by address register indirect with post-increment addressing, even if the comparison is only between a pair of addresses, and not a series. The address registers contain the addresses of the two locations to be compared.

Syntax : **CMPM.n (An)+,(An)+**

CMPI is used to compare an immediate value to an effective address, but not to an address register.

Syntax : **CMPI.n #$nnnn, ea**

Both of these forms of the CMP instruction can compare byte, word, and longword data.

DBEQ

Decrement and Branch until EQual. If the Z flag is clear, the branch is executed again. Thus, if the result of an operation is not zero, or a comparison does not prove the operands equal, the loop will be executed again. The loop ends when the count reaches -1, or when the Z flag is set.

e.g. The following program subtracts 1 from the low-byte contents of a data register. If the contents are not zero, the loop is executed again, decrementing the contents each time until the contents are zero. The loop count is specified as -1, so the loop will be executed 65536 times unless contents of the data register become zero.

```
10   ORIGIN
20   MOVE.L #$FFFFFFFF,D1       Loop counter.
30   MOVE.B #$aa,D0             Number compared to
                                0.
40   zero.loop                  Loop start.
50   SUBI.B  #$1,D0             Subtract 1
60   DBEQ  D1,zero.loop         If not 0, loop
                                again.
70   END
```

If the result of the subtraction is not zero, the Z flag will be cleared, and the loop will be executed again.

DBNE

Decrement and Branch until Not Equal. If the Z flag is set, this instruction will execute the loop again. Thus if the loop compares two values, the loop will continue to be executed if the values are the same, but will end if they are different. If the loop executes an arithmetic routine, the loop will continue to be executed if the result of the operation is zero, but will end with a non-zero result.

e.g. This program subtracts the contents of a data register from the contents of an address, and uses the result to divide into the contents of another data register. The loop ensures that if the result is zero, the loop is executed again and the contents of the next address are used for the subtraction, for maximum of 10 loops.

```
10   ORIGIN
20   MOVE.B   #$45,D0                  No divided into.
30   MOVE.B   #$10,D1                  Loop counter.
40   MOVEA.W   #$1000,A0               First address.
50   MOVE.B   #$12,D3                  Subtraction
                                       number.
60   SUBTRACT.LOOP                     Loop start.
70   SUB.W D3, (A0)+                   Subtraction.
80   DBNE D1,SUBTRACT.LOOP             If zero, loop
                                       again.
90   DIVS (A0)+,D0                     Divide.
100  END
```

The program assumes that the numbers involved in the subtraction will not produce a result which cannot be represented as a word data item. If the result of the subtraction is zero, the loop is repeated to ensure that the divisor operand is not zero, which would cause a 'division by zero' error at line 90.

DBF

Decrement and Branch until False. This is identical to the DBRA instruction, since it always executes the loop, and terminates by count only.

DBT

Decrement and Branch until True. This instruction always terminates. The loop is never executed again after this command.

DBGE

Decrement and Branch until Greater than or Equal to. The loop is terminated if either the N and V flags are both set, or if the N and V flags are both clear.

e.g. The following program adds #$1 to the contents of a data register each time the loop is executed. The contents are then compared to #$13. If the contents are greater than or equal to #$13, the loop ends.

```
10    ORIGIN
20    MOVE.B #$aaa,D1              Replace 'aaa'.
30    MOVE.B #$FF,D0               Loop count.
40    Compare.Loop                 Loop start.
50    ADDI.B #$1,D1                Add #$1.
60    CMPI #$13,D1                 Compare values.
70    DBGE D0,Compare.Loop         If not greater,
                                   loop.
80    END
```

The loop is executed #$20 (=32 decimal) times if the termination condition is not satisfied. Note that all instructions involving immediate values are suffixed with an 'I', to indicate that the source operand is an immediate value. By replacing 'aaa', the effect on the loop can be investigated.

The following loop instructions will have only explanations, no examples. Examples have not been included in order that the reader may experiment with the instructions, gaining valuable experiance in programming. The previous examples will act as guidelines to forming your own programs using loops.

DBGT

Decrement and Branch until Greater Than. This instruction is similar to the previous instruction, but ends only when the result of a comparison operation in the loop indicates that the destination operand in the comparison is greater than the source operand. For the loop to end, the Z flag must be clear, and the N and V flags must either be both set, or both clear.

DBLE

Decrement and Branch until Less than or Equal. The termination condition is that the destination operand in the comparison operation in the loop is less than or equal to the source operand. The condition specifies that the loop is ended if the N flag is set AND the V flag is clear, or if the N flag is clear AND the V flag is set. The loop is also terminated if the Z flag is set.

DBLT

Decrement and Branch if Less Than. This is similar to the previous instruction, but the loop is ended ONLY if the destination operand of the comparison is less than the source. The loop does not end if the operands are equal. The flag conditions are that the N flag is set AND the V flag is clear, or that the N flag is clear AND the V flag is clear.

The following two instructions assume UNSIGNED numbers.

DBHI

Decrement and Branch until HIgher than. This instruction is similar to DBGT, but assumes unsigned integers. The loop is terminated when the C flag AND Z flag are both clear.

DBLS

Decrement and Branch until Lower or Same. This instruction is similar to DBLE, terminating when the destination operand of a comparison in the loop is lower or the same as the source operand. Termination of the loop occurs either the C flag OR the Z flag are set.

DBMI

Decrement and Branch until MInus. If the result of the loop operations is a negative number, the loop is terminated. Thus, if the N flag is set, the loop is terminated.

Note that this instruction should NOT be used instead of the DBLT instruction to test the result of a comparison, since the N flag may be set after a positive result. An explanation of the circumstances in which this may occur is given in the previous chapter.

DBPL

Decrement and Branch until PLus. The loop is terminated when the result of the loop operations is a positive number, i.e. when the N flag is clear. The DBPL instruction should NOT be used instead of the DBGT instruction to test the result of a comparison, since the N flag may be clear after a negative result, as well as after a positive result. See previous chapter if an explanation is necessary.

DBVS

Decrement and Branch until oVerflow flag Set. If the result of the loop is larger than the data length specified, the V flag is set, and the loop is terminated. Thus, the loop terminates when the V flag is set.

DBVC

Decrement and Branch until V flag Clear. The loop terminates when the V flag is clear - i.e. when the result of the loop operations is within the data length expected. The termination condition is for the V flag to be clear.

Because the DBcc instructions use the same word length displacement method as the branch instructions, they are also restricted to a backwards displacement of 32768 bytes. To execute a loop in which the start of loop is more than 32768 bytes from the end of the loop, a JMP instruction is called by the DBcc instruction, to jump to the beginning of the loop.

The following sequence illustrates the way in which the JMP instruction is used.

```
Start.Loop                  Start of loop.
:                           Loop
:                           instructions
:                           here.
JMP next.but.two            JMP to avoid jump
                            back routine during
                            task execution.
Mid.Jump                    Jump back
JMP Start.Loop              routine.
NOP                         next.but.two address.
DBcc Dn, Mid.Jump           Branch to JMP instr.
:                           Rest of program
:                           instructions.
```

The JMP next.but.two is used to jump over the instructions to go to the beginning of the loop, since the loop would continue to infinity if they were executed at this point. Thus, the jump goes to the address of the NOP instruction, found as described in the previous chapter. The DBcc instruction is executed. If the loop is to be executed again, the DBcc instruction branches to the JMP instruction which sends program control back to the beginning of the loop.

The jump back is to executed when the loop is called, and a JMP is required to avoid the jump back instructions while the loop routine itself is being executed. If the jump back were not avoided, the program would always jump back to the beginning of the loop at that point.

Summary of termination status of flags

INSTRUCTION		FLAG STATUS				
		X	N	V	Z	C
DBCC		.	.	.	.	0
DBCS		.	.	.	.	1
DBEQ		.	.	.	1	.
DBNE		.	.	.	0	.
DBF		if false				
DBT		if true				
DBGE		.	1	1	.	.
	OR	.	0	0	.	.
DBGT		.	1	1	0	.
	OR	.	0	0	0	.
DBLE		.	1	0	.	.
	OR	.	0	1	.	.
	OR	.	.	.	1	.
DBLT		.	1	0	.	.
	OR	.	0	1	.	.
DBHI		.	.	.	0	0
DBLS		.	.	.	.	1
	OR	.	.	.	.	0
DBMI		.	1	.	.	.
DBPL		.	0	.	.	.
DBVS		.	.	1	.	.
DBVC		.	.	0	.	.

Chapter Six
MOVE and CMP

This is a short chapter, summarising the MOVE and CMP instructions. These instructions are the most frequently used instructions in assembly language, and each has several different forms, depending on the operands being used.

MOVE

The MOVE instruction is used to COPY a value into the destination operand. Note the use of the word COPY - the source operand, from which the value is copied, is UNCHANGED by the operation.

e.g. Suppose the data regster D0 contains the hex word number #$1234, and this is MOVEd into D1, with the instruction MOVE.W D0,D1. The data registers both contain #$1234 after the operation. Only D1 is changed by the operation.

This is characteristic of 68000 operations, since the result of an operation is usually stored in the destination operand.

The variations of the MOVE instruction are explained below. Note that no examples of use are given, since the MOVE instructions are used in every program in this book. They are best studied as part of a full program, to see their function.

Syntax : **MOVE ea, ea.**

The MOVE instruction usually expects one of the operands to be a data register.

Note that the source operand cannot be the status register, nor can the destination operand be an address register.

When data is moved into a data register, only the bits corresponding to the length of the data are affected.

e.g. If moving a byte into the data register, only the low-order byte is affected, being set to the value of the moved data. Word data only affects the low-order word. Longword data affects the entire contents, since data registers are longword data spaces. Data is not sign-extended when stored in a data register. To sign-extend a value in a data register, the **EXT** instruction must be used.

Syntax: **EXT.n Dn**

The EXT instruction is followed by the data register to be affected. The data length for the extension is specified as follows. To extend the low-order byte to a word, the instruction is EXT.W Dn. To extend the low-order word to a longword, the instruciton is EXT.L Dn. Thus, to extend a byte to a longword, the byte is first extended to a word, and then the word is extended to a longword. Note that *only data registers can be sign-extended* in this way. To store a byte sign-extended to a longword in memory, the byte value must be placed in a data register, sign-extended as described above, and then stored in memory.

MOVEA

MOVE to an Address register. The destination operand must be an address register. Only word and longword data may be stored in an address register. Word length data is automatically sign-extended to longword length.

The source operand may be any effective address.

MOVEQ

MOVE Quickly. This instruction is used for the fast copying of an immediate value, in the range 127 to -128 (decimal) to a data register. The data is automatically sign-extended to a longword when it is stored in the data register.

Syntax : **MOVEQ #$nn, Dn**

MOVE to CCR

MOVE to CCR. This moves a word length value into the low-order word of the status register - the Condition Codes Register. Thus the status flags in the CCR may be set to particular values.

Syntax : **MOVE ea, CCR**

These are the main MOVE instructions for the 68000. Other, more specialized, variations will be described when the conditions for their use are introduced.

Comparisons

The comparison instructions compare two values by subtracting the value in the source operand from that in the destination operand. Neither operand is affected by the comparison, since the result of the subtraction is discarded. The status flags are set according to the result, to indicate which of the operands was larger, or if they were equal.

CMP

CoMPare. The destination operand must be a data register, but the source operand can be any effective address, except an address register.

Syntax : **CMP.n ea,Dn**

CMPA

CoMPare to Address register. The destination must be an address register, but the source operand can be any effective address. Byte data may not be specified, and word data is sign-extended to longword data before comparison.

Syntax : **CMPA ea, An**

CMPI

CoMPare Immediate value. The source operand must be an immediate value, and the destination may be any effective address, except an address register.

Syntax : **CMPI.n #$nnn, Dn**

CMPM

CoMPare Memory locations. Both the source and destination operands are memory locations, and can only be addressed in post-increment addressing modes.

Syntax : **CMPM.n (An1)+, (An2)+**

See Chapters 4 and 8 for further details of this addressing mode.

Chapter Seven
Data Operations

Apart from the arithmetic operations already met in previous chapters, there are various other operations which may be performed on data in assembly language programs. Some of these have already been briefly discussed, as part of the section introducing binary numbers. These operations, AND, OR, NOT and the like, are known as *logical* operations, and are performed between two numbers, with the result being left in the destination operand.

For a more detailed explanation of the actual operation performed, refer to Chapter 2.

The logical operations are AND, OR, EOR, NOT, NEG.

AND

The AND instruction performs a bit-wise logical AND operation between the contents of the source and destination operands, leaving the result in the destination operand.

Syntax:	**AND ea, Dn**
or	**AND Dn,ea**

Data length: byte, word, longword.

Note that one of the operands must be a data register, and the destination operand cannot be an address register.

The AND operation is often used to find the remainder of a divison by a power of two, as explained in Chapter 2. The following program illustrates the use of the AND operation to find the remainder when $F3 is divided by $10 (i.e. 243 divided by 16 - decimal). The remainder is stored in D0, the destination operand.

```
10  ORIGIN
20  MOVE.B  #$F3, D0      Source operand
30  MOVE.B  #$F, D1       Destination operand
40  AND.B D1, D0          AND the operands.
50  END
```

Before: The low-order byte of D0 = 11110011, D1 = 00001111

After: D0=00000011

The result of the AND operation is stored in D0. Note that the value stored in D0 is ONE LESS than the divisor, i.e. $F instead of $10. To determine a remainder using this operation, the contents of the destination operand must be the divisor minus one.

This is method of determining a remainder is faster than using the DIVU and DIVS instructions, since the remainder is given directly. In the latter operations the remainder is stored as the high-word of a longword, and consequently a further operation, SWAP, is needed to access it.

This operation can only be used to find remainders for divisions in which the divisor is a power of two. The integer result of the divison cannot be found with this operation. To find the division result or a remainder for any other divisor, the DIVU or DIVS operations must be used.

ANDI

AND Immediate. This instruction is similar to the AND instructions, but the source operand is ALWAYS an immediate operand (numeric data), the destination operand can be any effective address.

Syntax: **ANDI #nnnn, ea**

Data length: byte, word, longword

Note: Most assemblers will accept AND with the above operands instead of ANDI, automatically making the distinction and substituting the correct code.

OR

This instruction performs a bit-wise OR operation on the source and destination operands, leaving the result in the destination operand.

Syntax: **OR ea, Dn**

or **OR Dn, ea**

Data length: byte, word, longword.

This operation is frequently used in graphics routines, to superimpose two characters on each other. The data for the characters is ORed, and the result is a character which is a combination of the two original characters. The OR instruction is not frequently used for any other function.

To illustrate the effect of OR, the following program performs the OR operation on the same numbers used in the AND example.

```
10   ORIGIN
20   MOVE.B  #$F3,  D0          Source  operand
30   MOVE.B  #$F,  D1           Destination  operand
40   OR.B D1,  D0               OR  the  operands.
50   END
```

Before: The low-order byte of D0 = 11110011, D1 = 00001111
After: D0=11111111

ORI

OR Immediate. This instruction is the same as OR, but always has an immediate value as the source operand.

Syntax: **ORI #nnnn, ea**

Data length: byte, word, longword.

Most assemblers will accept OR with the above operands instead of ORI, automatically adjusting the code according the operands.

EOR

Exclusive OR. This instruction performs a logical exclusive-or operation between the two operands, leaving the result in the destination operand.

Syntax: **EOR Dn, ea**

Data length: byte, word, longword.

Note that the source operand must be a data register.

To demonstrate the effect of the EOR instruction, the following program performs the EOR instruction on the same data used in the AND and OR examples.

```
10   ORIGIN
20   MOVE.B  #$F3,  D0          Source  operand
30   MOVE.B  #$F,  D1           Destination  operand
40   EOR.B  D1,  D0             EOR  the  operands.
50   END
```

Before: The low-order byte of D0 = 11110011, D1 = 00001111
After: D0=11111100

EORI

Exclusive-OR Immediate. This is the same as EOR, but the source operand must be an immediate value, not a data register.

Syntax: **EORI #nnnn, ea**

Data length: byte, word, longword.

Most assemblers will allow EOR to be used instead of EORI, adjusting the code produced according to the operands.

NOT

The NOT instruction performs a bit-wise logical NOT operation on the operand, leaving the result in the operand. This operation inverts all zeros in the binary contents to ones and vice versa. (i.e. it performs a ones complement operation on the number).

Syntax: **NOT ea**

Data length: byte, word, longword.

Thus, to form the twos complement (negative value) of a binary number, the following program could be used.

```
10   ORIGIN
20   MOVE.B  #$F3,  D0      Put value in D0.
30   NOT.B  D0              Invert for ones
                            complement.
40   ADDI.B   #$1,  D0      Add $1 to form twos
                            complement.
50   END
```

This forms the twos complement of the hex number $F3, leaving the result in D0.

NEG

NEGate binary. This instruction performs the operation of the above program, that is, it calculates the twos complement of the indicated number. Thus, it is used to convert a postive number to its negative value, and vice versa.

Syntax: **NEG ea**

Data length: byte, word, longword.

The following program examines a number to see whether it is negative or positive. If negative, the number is changed to positive. If positive, it is unaltered.

```
ORIGIN
MOVE.B #$F3, D0              Put value in D0.
BPL end                      If positive, end
NEG.B D0                     If negative,
                             change.
NOP                          Replace 'end' with
                             this address.
END
```

Replace 'end' with NOP address.

NEGX

NEGate eXtended binary number. This is used to form the twos complement of a binary number longer than 32 bits. As with other multiple precision instructions (ADDX, SUBX etc.) the Z-bit should be set and the X-bit clear before the operation is performed. The low-order longword is negated first, with the NEG instruction, and then the high-order part is negated with the NEGX instruction.

Syntax:	**NEGX ea**

Data length: byte, word, longword.

These instructions AND, OR, EOR, NOT and NEG are known as the logic instruction subset of 68000 instructions.

Shifts and Rotates

Another group of instructions which were also briefly discussed in Chapter 2 are the shifts and rotates. These instructions directly affect the entire contents of a memory location, as illustrated in Chapter 2.

Arithmetic Shifts - ASL

Arithmetic Shift Left. All the bits in the effective address are shifted to the left, placing the most significant bit into the carry and extend flags, and replacing the least significant bit with a zero.

Syntax:	**ASL Dn, Dn**
or	**ASL #n, Dn**
or	**ASL ea**

Data length: byte, word, longword

Shifts of memory addresses are always of word length, and are shifts of one bit. Shifts of data registers may be byte, word or longword, and may be shifts of 1 to 8 places, as specified by a constant or the contents of another data register.

The effective address ASL shifts the contents of the effective address left by one bit. The data register ASLs shift the contents of the destination data register left by the number of bits specified by the low-order nybble (4 bits) of the source operand. Thus, shifts of one to seven bits can be performed with only one instruction.

The following program shifts the low-order word of D0 left by two places.

```
10   ORIGIN
20   MOVE.L #$444321, D0
30   ASL.W #$2, D0              Shift left by two
                                bits
40   END
```

Before the shift: D0 = 00444321
After the shift: D0 = 00442100

Note that the high-order word of D0 is not affected, since only the low-order word is being shifted.

ASR

Arithmetic Shift Right. This performs a similar operation to ASL, but the shift is to the right instead of to the left. This operation assumes twos complement arithmetic.

Syntax:	**ASR Dn, Dn**
or	**ASR #n, Dn**
or	**ASR ea**

Data length: byte, word, longword.

The least-significant bit is placed in the carry and extend flags, and the most-significant is replaced by a zero or a one, depending on whether the number is positive or negative, respectively. This preserves the sign of the number.

As before, effective addresses (except data registers) can only be shifted by one bit.

Logical shifts

LSL

Logical Shift Left. This is identical to ASL.

Syntax:	**LSL Dn, Dn**
or	**LSL #n, Dn**
or	**LSL ea**

Data length: byte, word, longword.

Shifts of memory addresses are always of word length, and are shifts of one bit. Shifts of data registers may be byte, word or longword, and may be shifts of 1 to 8 places, as specified by a nybble-length constant or the low-order nybble contents of another data register.

LSR

Logical Shift Right. All bits are shifted right, as with ASR, but twos complement arithmetic is not assumed, and the most-significant bits are replaced with zeros, regardless of the sign of the number.

Syntax:	**LSR Dn, Dn**
or	**LSR #n, Dn**
or	**LSR ea**

The least-significant bit is placed in the carry and extend flags.

Rotates

These are similar to the shifts, but no bits of the operand being shifted are lost.

ROL

ROtate Left. The bits of the operand are rotated left. The most-significant bit is transferred to the least-significant position, and also copied into the carry flag.

Syntax:	**ROL Dn, Dn**
or	**ROL #n, Dn**
or	**ROL ea**

Data length: byte, word, longword.

As with the shifts, memory addresses can only be rotated one bit at a time, and only a word may be rotated. Data registers may be rotated by up to eight bits, and the rotate may be of a byte, a word or a longword.

The following program rotates the low-order word of D0 left by two places.

```
10   ORIGIN
20   MOVE   #$444321,   D0
30   ROL.W   #$2,   D0
40   END
```

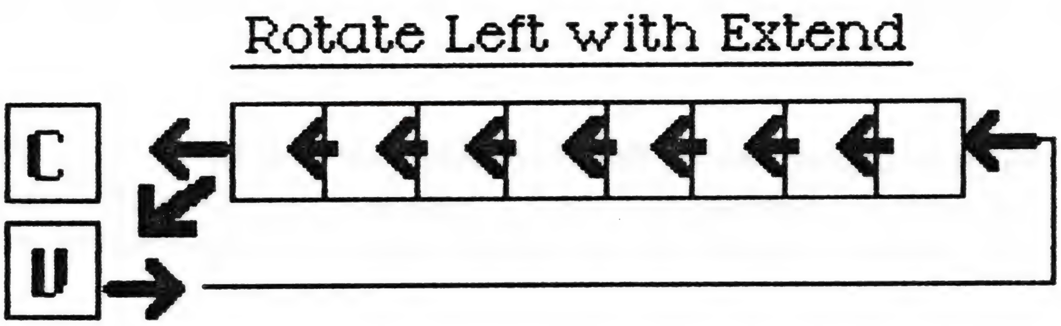
Before the ROL instruction: D0 = 00444321, (= 0100001100100001 (bin))
After the ROL instruction: D0 = 00443214, (= 0011001000010100 (bin))
Note that the high-order word of D0 is unchanged, since the rotate is specified to be performed on the low-order word only.

ROXL

ROtate with eXtend, Left. This instruction is similar to ROL, but the most-significant bit is copied into the extend flag and the carry flag, and the original contents of the extend flag are copied into the least-significant bit.

Syntax:	**ROXL Dn, Dn**
or	**ROXL #n, Dn**
or	**ROXL ea**

Data length: byte, word, longword.



The following program performs an ROXL on a memory address.

```
10  ORIGIN
20  MOVE.W  #$1FF,  $1000   $1FF  =  0111111111
                            (bin).
30  ROXL  $1000             Rotate.
40  END
```

Before rotation:	$1000 = 0111111111 (binary), and X flag = 0
After rotation:	$1000 = 111111110 (binary), and X flag = 1

ROR

ROtate right. All the bits in the operand are shifted right. The least-significant bit is placed into the carry flag and also in the most-significant bit position.

Syntax:	**ROR Dn, Dn**
or	**ROR #n, Dn**
or	**ROR ea.**

Data length: byte, word, longword.

Note that memory addresses can only be rotated by one bit, and only word length contents can be rotated. Data registers can be rotated by up to eight bits, and any data length contents can be rotated.

The following program rotates the contents of D1 right by 4 bits.

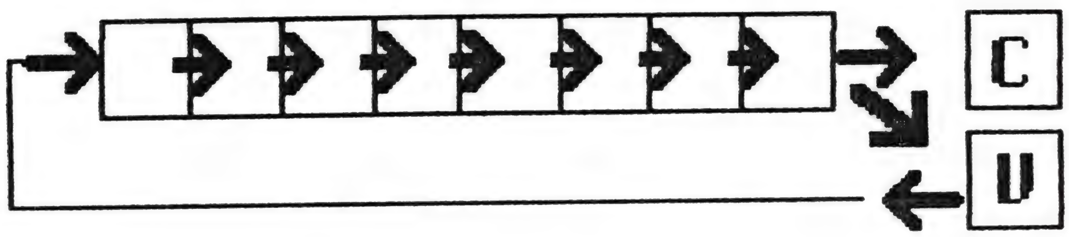
```
10   ORIGIN
20   MOVE  #$444321,  D1
30   ROR  #$4,  D1              Rotate  right  by  4.
40   END
```

Before the ROR instruction:	D1 = 00444321, (= 0100001100100001 (bin))
After the ROR instruction:	D1 = 00441432, (= 0001010000110010 (bin))

ROXR

ROtate with eXtend, Right. ROXR is similar to ROXL, but shifts all the bits right, placing the least-significant bit in both the carry and extend flags, and placing the previous contents of the extend flag in the most-significant bit position.

Rotate Right with Extend



Syntax: **ROXR Dn, Dn**
or **ROXR #n, Dn**
or **ROXR ea**

Data length: byte, word, length.

Note that the restrictions on rotation of memory addresses apply.
The following program performs an ROXR on a memory address.

```
10   ORIGIN
20   MOVE.W #$1FF, $1000    $1FF = [illegible]
                            (bin).
30   ROXR  $1000            Rotate.
40   END
```

Before rotation:	$1000 = 0111111111 (binary), and X flag = 0
After rotation:	$1000 = 011111111 (binary), and X flag = 1

Data manipulation.

The following set of instructions do not perform any arithmetic functions, and are used to rearrange data within its storage area. These instructions may be used to arrange data so that it can be more easily accessed by an instruction, or to make data suitable for a particular instruction.

EXG

EXchanGe. This instruction exchanges the entire 32 bit contents of two registers. The registers may be both data registers, address registers, or one of each.

Syntax: **EXG Rn, Rn**

Data length: longword.

The data length does not need to be specified, since there is only one possible value.

The instruction may be used principally to alter the contents of address registers, or to exchange the contents of two address register, for reasons which will be explained in Chapter 10.

The following program uses the EXG instruction to exchange the contents of A0 and D0, in a loop, altering the address each time.

```
10    ORIGIN
20    MOVE.L #$444321, A0
30    MOVE.L #$111111, D0
40    MOVE.B #$10, D1              Loop count
50    loopstart
60    MOVE.B #$10, (A0)            Put in addr.
                                   'pointed to'
70    EXG D0, A0                   Exhange contents.
80    ADD.L #$10, D0               Increment data in
                                   D0.
90    EXG DO, AO                   Exhange again.
100   DBRA D1, loopstart           Loop if count not
                                   -1.
110   END
```

The program stores the immediate value #$10 in memory locations $10 addresses apart, starting at $444321. The register contents are exchanged twice, once to alter the contents of A0, and the second time to place the new value in A0.

Note that 'loopstart' does NOT have to be replaced by an address, since it is defined before it is used.

EXT

EXTend. This instruction sign-extends the contents of a data register, to either word or longword length. The sign bit of the number is copied into all the remaining bits in the register to extend the length of the operand.

Syntax: **EXT Dn**

Data length: word, longword.

The following program sign-extends the contents of D0 and D1. D0 contains a byte length data item, to be extended to word length. D1 contains a word length data item to be extended to longword length.

```
10    ORIGIN
20    MOVE.B  #$10,  D0
30    MOVE.W  #$1234,  D1
40    EXT.W  D0                    Extend  D0  to  a  word
50    EXT.L  D1                    Extend  D1  to
                                   longword.
60    END
```

```
Before extension:     D0= -------------------------00010000
                                     (bin)
                      D1= ----------------0001001000110100
                                     (bin)
After extension:      D0= 00000000000000000000000000010000
                                     (bin)
                      D1= 00000000000000000001001000110100
                                     (bin)
```

SWAP

This instruction exchanges the high-order and low-order words of a data register. This instruction is particularly useful in conjunction with the division instructions (see Chapter 3) where the result of the operation is stored in the low-order word of the data register, and the remainder is stored in the high-order word. The SWAP instruction is needed to access the remainder after the operation.

Syntax: **SWAP Dn**

Data length: word.

No data length needs to be specified, since there is only one possible value.

The following program exchanges the upper and lower words of D0.

```
10   ORIGIN
20   MOVE  #$444321,  D0
30   SWAP  D0
40   END
```

```
Before the SWAP instruction:        D0 = 00444321
After the SWAP instruction:         D0 = 43210044
```

CLR

CLeaR. This instruction resets all the specified bits of the effective address to zero.

Syntax: **CLR ea**

Data length: byte, word, longword.

The following program clears the low-order byte of D0, the low-order word of D1, and a longword starting at memory address $1000.

```
10   ORIGIN
20   MOVE  #$444321,  D0
30   MOVE  #$444321,  D1
40   MOVE  #$12344567,  $1000
50   CLR.B  D0
60   CLR.W  D1
70   CLR.L  $1000
80   END
```

Before CLR: `D0 = 00444321, D1 = 00444321, $1000 = #$12344567`

After CLR: `D0 = 00444300, D1 = 00440000, $1000 = #$00000000`

TST

TeST. This instruction compares the operand to zero. The CCR flags are set according to the result. This instruction is the same as CMP.n #$0, ea.

Syntax: **TST ea**

Data length: byte, word, longword.

Bit-manipulation

An important part of assembly language is manipulation of individual bits, using non-arithmetic operations. One form of bit-manipulation, called bit-testing, has already been introduced in the branch and loop instructions. Bit-testing checks the value of a particular bit, but does not alter the value of that bit. The branch and loop instructions only check the bits in the CCR, but bit-testing can be performed on any data bit stored in memory.

The instruction used for individual bit-testing is **BTST**.

Syntax : **BTST Dn, ea**

or **BTST #nnnn, ea**

The effective address is the address in which the bit to be tested is stored, and may be either a memory address, or a data register, but not an address register. The number of the bit to be tested is stored either in a specified data register, or given directly in the instruction. If the effective address is a memory address, the bit number must be in the range 0-7, since a single address contains only one byte. If the effective address is a data register, the bit number can be in the range 0-31, since a data register contains longword data. No data length is specified with the instruction.

When the processor executes a BTST instruction, the specified bit in the effective address is tested, and the Z-flag in the CCR is set to the same value as the bit. Thus, if the bit is clear, the Z-flag will be cleared. If it is set, the Z-flag will be set.

The result of the test can be checked with a BNE instruction, as follows.

```
10   ORIGIN
20   MOVE  #$1234,  D0
30   BTST  #5,  DO                Test  5th  bit  of  D0.
40   BNE  end                     If  bit=0,  end.
50   ADD  #$23,  DO               If  bit=1,  add  $23
60   NOP                          Substitute  for
                                  'end'.
70   END
```

Replace 'end' with NOP address, as before. Note that the BTST instruction considers the zeroth bit to be the least-significant bit of the address, i.e. the rightmost bit.

By changing line 20, placing different values in D0, and investigating the contents of D0 after the operation, the value of the fifth bit can be deduced in each case.

BCHG

Bit test and CHanGe. This instruction tests the specified bit, and sets or clears the Z flag according to the state of the bit. The bit is then changed to have to opposite value.

e.g. If the bit is zero (clear), the Z flag is cleared, and the bit is changed to one (set).

If the bit is one (set), the Z flag is set, and the bit is changed to zero (clear).

Syntax: **BCHG Dn, ea**

or **BCHG #n, ea**

Data length: byte or longword.

If the destination is a data register, bits 0-31 can be tested. If it is a memory address, bits 0-7 may be tested. The number of the bit position in the effective address to be tested is given in the instruction, either indirectly (stored in a data register), or as an immediate value. The bit number is modulo numbering, according to the type of destination operand. For a data register the numbering is modulo 32, and for memory addresses the numbering is modulo 8. Modulo numbering implies that, if the number given is larger than the range allowed, the numbers is divided by the modulo number until the remainder is within the specified range. This remainder is then used as the bit position number.

The following program tests the 5th bit in data register D0, and changes it.

```
10   ORIGIN
20   MOVE.B  #$10,  D0
30   BCHG  #$5,  D0          Test  and  change.
40   END
```

Before the BCHG instruction: `D0 = 00010000, Z flag = 0`
After the BCHG instruction: `D0 = 00000000, Z flag = 1`

BCLR

Bit test and CLeaR. The specified bit is tested, and the Z flag set or cleared according to the state of the bit. The bit is then cleared.

Syntax:	**BCLR Dn, ea**
or	**BCLR #n, ea**

Data length: byte or longword.

As with BCHG, modulo numbering is used to determine the bit position. The same restrictions apply to the range of bit positions which can be tested, according to the type of destination operand.

The following program tests and clears the 4th bit in the contents of memory address $1000.

```
10   ORIGIN
20   MOVE.B  #$F,  $1000
30   BCLR  #$4,  $1000         Test  and  clear.
40   END
```

Before the BCLR instruction: `$1000 = 00001111, Z flag = 0`
After the BCLR instruction: `$1000 = 00000111, Z flag = 1`

BSET

Bit test and SET. This instruction is similar to the previous instruction, but sets the bit, rather than clearing it.

Syntax:	**BSET Dn, ea**
or	**BCLR #n, ea**

Data length: byte or longword.

The same restrictions on data length apply as before.

Using the same example as for BCLR, the effect of BSET can be shown.

```
10   ORIGIN
20   MOVE.B  #$F,  $1000
30   BSET  #$4,  $1000        Test  and  set.
40   END
```

Before the BSET instruction: `$1000 = 00001111, Z flag = 0`
After the BSET instruction: `$1000 = 00001111, Z flag = 1`

The Z flag is set or cleared according to the state of the tested bit. Since this instruction sets the bit after testing, there is no effect on the contents of D0, since the specified bit was already set.

Useful for Saving Memory

The bit-testing instructions are very useful for saving memory. If a program has several variables which may take one of two values - to indicate whether or not an event has occurred - each variable can be represented as one bit in a memory location byte. To determine whether or not the event has occurred, the particular bit is tested with a bit-test instruction. This method of storing 'event-indicators' also has the advantage that all the variables are in one place, and are less easy to lose track of. Large numbers of variables in a program can lead to problems remembering where each is stored if a careful note is not made.

TAS

Test And Set. This instruction tests the most-significant bit of the operand. If the msb is set, the N flag is set. If it is clear, the Z flag is set. After the test, the msb is always set.

Syntax: **TAS ea**

Data length: byte.

Since the data length can only be one value, no length needs to be specified.
The following program tests the msb of $1000, and sets it.

```
10   ORIGIN
20   MOVE.W  #$1234,  $1000
30   TAS  $1000                 Test  msb  and  set.
40   END
```

Before the TAS instruction:
```
$1000  = 0001001000110100
Z flag = 0
N flag = 0
```

```
After the TAS instruction:   $1000  = 1001001000110100
                             Z flag = 1
                             N flag = 0
```

The Z flag is set because the msb was initially clear. The N flag is not set for the same reason.

Byte testing

There is a set of instructions within the 68000 set which is used exclusively to test and set bytes according to specified condition codes. These conditions are the same as those for branches (the Bcc instructions) and thus will be only briefly covered here. For a further explanation of each condition, see Chapter 4.

All the instructions are in the form Scc, and the 'cc' is replaced by an appropriate condition code.

The byte defined as the destination is set to $FF if the condition is satisfied, otherwise it is cleared to zero.

Syntax : **Scc ea**

Data length : byte.

The data length does not need to be specified, since it can only be a byte.

SCC	Set if Carry Clear.
SCS	Set if Carry Set.
SEQ	Set if EQual.
SF	Set if False.
SGE	Set if Greater than or Equal.
SGT	Set if Greater Than.
SHI	Set if HIgher than.
SLE	Set if Less than or Equal.
SLS	Set if Less than or Same.
SLT	Set if Less Than.
SMI	Set if MInus.
SNE	Set if Not Equal.
SMI	Set if PLus.
ST	Set if True.
SVS	Set if oVerfow Set.
SVC	Set if oVerflow Clear.

Chapter Eight Addressing modes

Addressing modes are used to access the different locations in which data can be stored in the computer. Most instructions in assembly language can be used to refer to a variety of storage areas, e.g. data registers, address registers, memory locations and the like. In order to avoid having a large number of different instructions all performing the same task, but on data in different types of storage area, addressing modes are used. Different addressing modes *address*, or refer to, different memory storage areas. They indicate whether the data is to be found in a data register, a memory location, or if it is an immediate value, etc.

Several addressing modes have already been discussed earlier in the book, when they were required by an instruction. Some instructions require a specific addressing mode to be used, since they operate on data stored in only one type of storage, but most will accept data from a variety of storages and thus support several addressing modes.

Addressing modes can be used for operations other than merely indicating the type of data being used. These other operations include automatically decrementing or incrementing the contents of the address registers. Such addressing modes are useful in some loops, since they decrease the number of instructions required to execute the loop, if it involves changing the contents of an address register.

The addressing mode being used is represented in the six least significant bits of the instruction word. The 68000 stores instructions as a series of words (from one to five words) in memory. The first word, the *instruction* word, specifies the operation to be performed, the length of the operation (byte, word etc.). The remaining *extension* words give the operation data. The addressing mode being used forms part of the instruction word.

There are fourteen addressing modes available in the 68000 instruction set. These addressing modes can be divided into three groups, depending on the form in which the instruction specifies the operand.

Note: The operand referred to by the addressing mode is the source operand.

(i) Register specific. A specific register (data or address) is referred to, and is identified by the three least-significant bits of the operation word.

(ii) Implicit reference. The instruction does not need any location specified, since only one specific location can be referred to by the instruction. For instance, the instruction PEA pushes the effective address indicated onto the stack, a storage area in memory. The stack does not have to be specified as the storage area, since it is implied by the operation.

(iii) Effective addressing. The operand is identified by the contents of an effective address field within the instruction word. An 'effective address' is a general description which covers several types of operand - immediate data, memory, registers. A 'field' is a section of the word. In this case, it is bits 0-5.

These bits specify the operand as follows :-

Bits 3-5: These specify the addressing mode to be used.

Bits 0-2: These specify the register or location being accessed.

Effective addressing is the most frequently used type of addressing mode, and has three sub-modes:

(i) Register direct. Bits 0-2 specify a data register or an address register. Bits 3-5 indicate that the data is held in the specified register.

(ii) Memory addressing. Bits 0-2 specify the address register in which the address of the location is stored. The data is fetched from the location 'pointed to' by the address register. Bits 3-5 indicate that the data is in the location, not the address register.

(iii) Special addressing. A code indicating the special mode being used is stored in bits 0-2.

The rest of this chapter is a detailed description of each addressing mode, together with examples of use. At the end is a short summary of each addressing mode, and a list of the addressing modes, each followed by the instructions which may use that mode.

Register direct modes.

(i) Data register direct.

The data to be used is stored in the specified data register.

[data register] = [data store]

Syntax: **Dn**

e.g. MOVE.B D5, D4 copies the low-order byte of D5 into the low-order byte of D4.

(ii) Address register direct.

The data is stored in the address register specified.

[address register] = [data store]

Syntax: **An**

e.g. MOVE.W A2, D4 copies the low-order word of A2 into the low-order word of D4.

Memory addressing modes

(iii) Address register indirect.

The data is stored in the location whose address is stored in the address register specified. The data is said to be 'pointed to' by the address register.

[address register] ---> [data store]

Syntax: **(An)**

e.g. MOVE.B (A0), D4 copies the low-order byte of the location 'pointed to' by A0 into the low-order byte of D4.

(iv) Address register indirect with post-increment.

This is similar to (iii), but after the operation has been executed, the contents of the specified address register are incremented by 1, 2, or 4, depending on whether the operation was executed on a byte, a word, or a longword, respectively.

[address register] ---> [data store] ..operation.. [inc. address register]

Syntax : **(An)+**

e.g. CMP.L (A3)+, D2 compares the operands and then increments the contents of A3 by 4, since this was a longword operation. Before the operation is executed, A3 = nnnn. After the operation, A3 = nnnn+4.

Note: If A7 is being used to store the address, the register is incremented by 2 after byte operations.

(v) Address register indirect with pre-decrement.

This is similar to (iv), but the address register contents are *decremented before* the operation is executed. The contents are decremented by 1, 2, or 4 depending on the length of the data for the operation.

[address register]-[decrement] ---> [data store]

Syntax: **-(An)**

e.g. CLR.B -(A4) clears the low-order byte of the address 'pointed to' by A4, after the contents of A4 have been decreased by 1.

These two addressing modes, (iv) and (v) are very useful in loops, since the loop can perform the required task on a consecutive series of locations, without any arithmetic operations being used to alter the contents of the address register.

For instance, in a loop with a (An)+ statement the contents of the address register are automatically incremented after the operation, thus giving the next location to be used, without needing any further instructions.

(vi) Address register indirect with offset.

The data is stored in the memory location whose address is formed by the contents of the specified address register and a constant (the offset), which is a 16-bit sign-extended integer.

[address register]+[offset] ---> [data store]

Syntax: **+c(An)** or **-c(An)**

'c' is the offset, and may be positive or negative, as indicated.

e.g. ADD.B -2(A1), $1000 adds together the low-order byte of the contents of the address 'pointed to' by subtracting #2 from the contents of A1 and the low-order byte of the contents of address $1000.

Note: The contents of the address register are NOT altered by this operation. The offset is added to the value of the contents during the operation, but the contents are not altered. Thus, in the above example, if A1 contains $1234 then the address 'pointed to' is $1232, but the contents of A1 remain $1234.

(vii) Address register indirect with offset and index.

The address 'pointed to' is the sum of the contents of the address register, the least-significant byte of an offset, and the contents of an *index register.*

[address register]+[index register]+[offset] ---> [data store]

Syntax: **c(An, Rn)**

'c' is the offset, and may be positive or negative.

e.g. MOVE.L 3(A2, D6.L), D3 copies into the whole of D3 the contents of the address found by adding the contents of A2, the contents of D6, and the offset 3.

The 'L' after D6 indicates that the index is a longword. If no length is specified, the index is assumed to be a word, and is sign-extended to a longword.

Special address modes

(viii) Absolute short address.
The data is stored in the memory address specified.
[address] = [data store]
Syntax: **$nnnn**
$nnnn is a word, sign-extended to a longword.
e.g. ADDA.W $1200, D1 adds the word contents stored in the two bytes $1200 and $1201 to the contents of D1.

(ix) Absolute long address.
The data is stored in the memory address specified.
[address] = [data store]
Syntax: **$nnnnnnnn**
e.g. ADDA.L $FF1F23, D5 adds the longword data stored in addresses $FF1F23 - $FF1F26 to the contents of D5.

The difference between short and long absolute addressing is that, while absolute long addressing can address the whole range of memory, absolute short addressing can only address the top 32K and bottom 32K of memory. Since the absolute short mode sign-extends the address before use, addresses $8000-$FFFF refer to addresses $FFFF8000 and above, and thus access the top 32K. Addresses $0000-$7FFF refer to the bottom 32K.

(x) Program counter relative with offset.
The address required is found by adding a sign-extended 16-bit integer to the contents of the program counter.
[p.c.]+[offset] = [address of next instruction to be executed]
Syntax: ***c**
'c' is the offset.
e.g. BSR *-17 subtracts 17 from the contents of the program counter, and program control passes to the subroutine beginning at that address.

(xi) Program counter relative with offset and index.
This is similar to (x), but the contents of an index register are also added to the program counter.
[p.c.]+[index register]+[offset] ---> [data store]
Syntax: **c(P.C, index reg.)**
'c' is the offset, and may be positive or negative.
e.g. MOVE.B $23(P.C, D6.L), D1 adds together the contents of the program

counter and the longword contents of D6, together with the offset ($23), and stores the first byte of the resulting addrε ss in D1.

Note: The index register contents are word-length and sign-extended by default. Other lengths must be specified, as above.

(xii) Immediate data.

The data is given in the instruction.

Syntax: **#nnnn** (decimal)

If the data is hex, the syntax is **#$nnnn.**

e.g. MOVE.B #$12, D1 moves the byte value $12 into D1.

(xiii) Implicit reference.

An implicit instruction is one which requires no operands, since the operand is implied by the question.

e.g. RTS returns control from a subroutine to the address from which the address was called. The address is stored in a section of memory called the stack. When the RTS instruction is found, the processor goes to the stack to fetch the address. The stack is implied by the instruction as the operand from which the operation fetches the address. Thus, it does not need to be stated, since there is nowhere else that the address could be fetched from.

(xiv) Status Register Addressing.

This addressing mode is used to indicate that the instruction will be carried out on the status register. If the instruction specifies a byte length operation, then the operation will affect only the user byte of the status register. The user byte is also called the condition code register. Word length operations affect both user and system bytes.

Syntax: **SR** (status register)
or **CCR** (user byte - byte operations only)

e.g. ORI #$4, CCR sets the zero flag in the CCR.

Instructions using the SR mode are privileged instructions, and may only be used under special conditions. Privileged instructions are discussed in Chapter 13.

Summary

The following summary of the addressing modes includes the contents of the mode and register fields of the instruction. These are bits 0-5 of the first instruction word.

Addressing mode	**Syntax**	**Mode**	**Register**
Data register direct	Dn	000	Reg.No.
Address register direct	An	001	Reg.No.
Address register indirect	(An)	010	Reg.No.
Addr.reg.indirect with post.incr.	(An)+	011	Reg.No.
Addr.reg.indirect with pre.decr.	-(An)	100	Reg.No.
Addr.reg.indirect with offset	c(An)	101	Reg.No.
Addr.reg.indirect with offs. & index	c(An,Rn)	110	Reg.No.
Absolute short	$nnnn	111	000
Absolute long	$nnnnnn	111	001
PC relative with offset	c(PC)	111	010
PC relative with offset & index	c(PC,Rn)	111	011
Immediate	#($)nnnn	111	100
Status register (privileged)	SR	111	100
Condition codes register	CCR	111	100

Use of addressing modes

The following table lists each instruction, and the addressing modes it supports.

Note: Not all of the instructions in this list have been discussed to this point in this book. These instructions are marked with an asterisk(*).

Instruction	**Addressing modes (by number)**
*ABCD	(i),(v)
ADD (ea=source)	(i),(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x) (xi),(xii)
(ea=destination)	(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
ADDA	As ADD (ea=source)
ADDI	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
ADDQ	(i),(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix)
ADDX	(i),(v)
AND (ea=source)	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
(ea=destination)	(iii),(iv),(v),(vi),(vii),(viii),(ix)

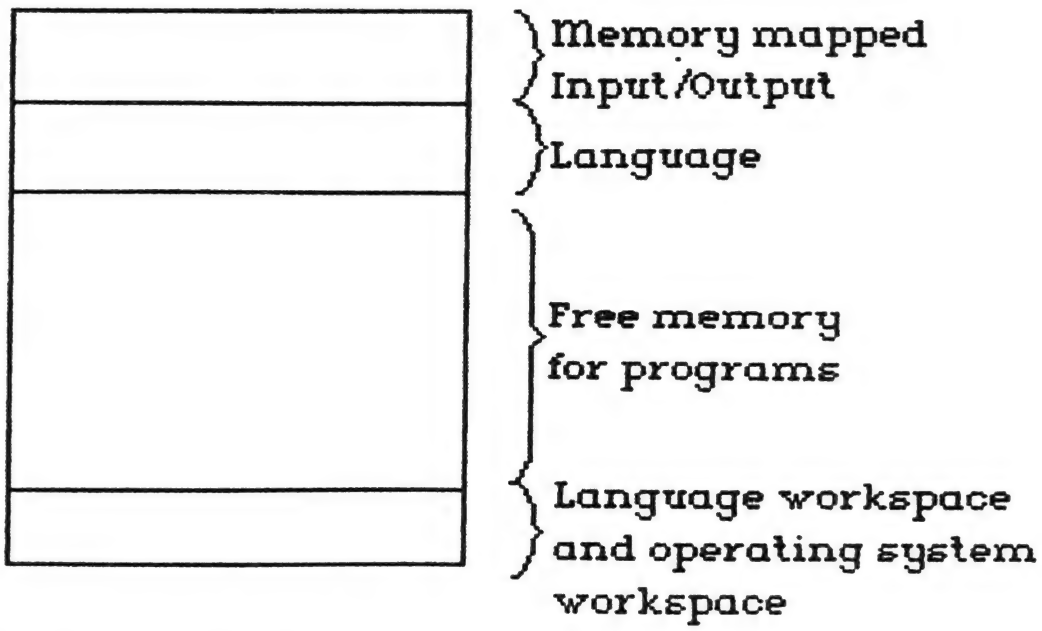
ANDI	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(xiii) (xiv)
ASL	(iii),(iv),(v),(vi),(vii),(viii),(ix)
ASR	As ASL
BCHG	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
BCLR	As BCHG
BSET	As BCHG
BTST (bit no. in Dn)	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
(bit no.=immed.)	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x);(xi)
*CHK	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
CLR	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
CMP	(i),(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x) (xi),(xii)
CMPA	As CMP
CMPI	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
DIVS	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
DIVU	As DIVS
EOR	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
EORI	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(xiii)
JMP	(i),(iii),(vi),(vii),(viii),(ix),(x),(xi)
JSR	(iii),(vi),(vii),(viii),(ix),(x),(xi)
*LEA	(iii),(vi),(vii),(viii),(ix),(x),(xi)
LSL	(iii),(iv),(v),(vi),(vii),(viii),(ix)
LSR	As LSL
MOVE (ea=source)	(i),(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x) (xi),(xii)
(ea=destination)	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
*MOVE to CCR	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
*MOVE to SR (priveliged)	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x) (xi),(xii)
MOVEA	As MOVE (source)
MOVEM (mem.=dest)	(iii),(v),(vi),(vii),(viii),(ix)
(mem.=source)	(iii),(iv),(vi),(vii),(viii),(ix),(x),(xi)
MULS	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
MULU	As MULS

*NBCD	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
NEG	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
NEGX	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
NOT	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
OR (ea=source)	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi) (xii)
(ea=destination)	(iii),(iv),(v),(vi),(vii),(viii),(ix)
ORI	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix),(xiii) (xiv)
*PEA	(iii),(vi),(vii),(viii),(ix),(x),(xi)
ROL	(iii),(iv),(v),(vi),(vii),(viii),(ix)
ROR	As ROL
ROXL	As ROL
ROXR	As ROL
*SBCD	(i),(v)
Scc	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
SUB	(iii),(iv),(v),(vi),(vii),(viii),(ix)
SUBA	(i),(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x) (xi),(xii)
SUBI	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
SUBQ	(i),(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix)
SUBX	(i),(v)
TAS	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)
TST	(i),(iii),(iv),(v),(vi),(vii),(viii),(ix)

Any instructions, such as branches, which are not included in this summary have their own addressing mode, and cannot use any other. Also not included are implicit instructions ·vhich have no operands.

Chapter Nine
The Stack and other Memory Areas

The memory of a computer is used for many different purposes - to store programs, to store variables, to store temporary values in calculations, and to store data for a program. The operating system of the computer uses memory to store data with which it is working. All these memory uses involve a type of memory called RAM (Random Access Memory), which can be altered by a user, unlike ROM (Read Only Memory). ROM is used to store permanent programs which are always needed by the computer - for instance, the program which displays the disc and query mark in the middle of a Macintosh screen when the computer is powered up is held in memory. ROM cannot be altered by a program, nor is any program in it wiped when power is shut off. RAM, however, loses all its contents when power is switched off. This is why programs must always be saved onto disc before switching a computer off, otherwise the work will be lost.



Memory mapped Input/Output
Language
Free memory for programs
Language workspace and operating system workspace

Example of a RAM memory map

The memory of the computer performs many functions. In order that these functions are all carried out smoothly, with no two tasks requiring the same memory, the operating system of the computer reserves sections of memory for its own use in calculations. These areas of memory are known as the *operating system workspaces*. The operating system uses them to store the intermediate results of calculations.

Other areas of memory also reserved are used as input and output areas. Data which is to sent to the screen, or to a printer or disc drive is stored in locations in memory which correspond to, for instance, a particular screen location. By storing the data in that memory location, the processor causes the data to be displayed on the screen. These areas are known as *memory-mapped input/output*. These operating system workspaces and input-output areas can be manipulated by programmers, but always with extreme caution, since strange results can occur if care is not taken when altering these areas.

Another use of memory is for *buffers*, so called because they act as buffers between the processor and the peripherals - screen etc. Buffers are required because the processor executes instructions very much faster than the majority of peripherals. So that the processor does not need to wait while the peripheral finishes executing an instruction before sending the next one, the processor stores a number of instructions in a buffer and then continues with another task while the peripheral processes the information in the buffer. As the buffer is emptied, the processor sends another batch of data for processing.

A keyboard buffer acts in the opposite way to that described above. It stores information coming into the computer, so that the processor is not forever checking the keyboard, which would slow it down considerably. Instead, input data is stored in the keyboard buffer, and the processor checks the contents of the buffer at less frequent intervals than would otherwise be necessary. This frees the processor to carry out other tasks instead of having to stop and check the keyboard. Like workspaces, buffers can also be manipulated by programmers.

This is particularly useful with buffers such as the keyboard buffer. When giving a choice of options in a program, pressing the <ENTER> key alone could select the default option, whose symbol is then inserted into the keyboard buffer as if it had been typed into the machine.

Since all these operations and functions change the contents of memory, they need to use RAM for storage areas. This is why the entire contents of RAM are never all at the programmers disposal.

When using languages such as BASIC, which require workspace for the translation of the program as it is running, the operating system of the language reserves further sections of memory for the language workspace. However, in assembly language, only the workspace essential to the operating system is set aside, and these areas may be manipulated by the programmer, although peculiar effects may occur if care is not taken.

Of the areas set up by the processor, the most often used by programmers is the hardware stack. Stacks are areas of memory used as temporary stores, for data which is being used for calculations. It is, in effect, an electronic 'scribbling pad'. Stacks may be set up by a programmer in any section of free memory, but the hardware stack is automatically created by the 68000 on power-up.

The stack is known as a last-in-first-out (LIFO) structure. The last piece of information stored is the first to be taken out. The address of the top of the stack (where the most recent data is stored) is stored in an address register known as the stack pointer. This register is A7 for the hardware stack, but a stack created by the programmer may have any address register as its pointer. For the hardware stack most assemblers will accept SP instead of A7 to indicate the stack pointer. The stack pointer always holds the address where the next data will be stored. Each time another piece of data is added to the stack the stack pointer is altered, and is either incremented or decremented, depending on whether you wish the stack to grow upwards or downwards in memory.

Thus, the addressing modes used with the stack are pre-decrement or post-increment. These automatically update the stack pointer whenever any data is accessed or added to the stack. For a downwards-growing stack, the pre-decrement mode -(An), where An is the address register being used as the stack pointer, is used to calculate the new top of the stack and then push data onto the stack. To remove , or 'pop', data from the stack, the indirect memory addressing mode (An) is used, if the data is on the top of the stack. If it is stored lower down the stack, the value of the address where it is stored must first be calculated and stored in the stack pointer before 'popping' the data. With upwards-growing stacks, the procedure is the same, but the post-increment mode, (An)+, is used to push data onto the stack.

e.g. Suppose the stack originally looks like this :

Address	**Contents**
10000	1234
09998	5678
09996	3456
09994	1289

The stack pointer contains the value 09994. The instruction MOVE #1234,-(SP) would cause the stack to be extended downwards:

Address	**Contents**
10000	1234
09998	5678
09996	3456
09994	1289
09992	1234

The stack pointer now contains 09992. The pointer is decreased by two each time because the data is word-length, two bytes.

Data on the stack is always either word or longword length. Byte length data is sign-extended before storage.

LINK

Using the stack as described can be an inefficient use of memory, since the stack continues to grow each time data is added to the stack. A more efficient use of the stack memory is achieved with the use of the LINK instruction. This instruction temporarily allocates an area of the stack to be used to store data. This is called a stack frame. After all calculations using the stored data have been carried out, the frame is deallocated with the UNLK instruction.

The LINK instruction takes two operands, an address register and a word length frame size operand.

Syntax : **LINK An,#nnnn**

Data is word length.

The contents of the address register are pushed onto the stack, and the value of the stack pointer is stored in the address register. The frame size operand is added to the stack pointer to allocate memory for the variables.

Note: The frame size operand is negative for a stack that grows downwards in memory.

The storage area formed is addressed using displacements relative to the contents of the address register which thus acts as the frame pointer. These displacements should be negative for a downwards growing stack.

UNLK deallocates the frame, copies the contents of the address register into the stack pointer, and 'pops' the original contents of the address register from the stack.

By using this method of reserving stack space to be reusable memory can be reused and made more easily accessible to a program. The memory saving aspect is of particular importance in large programs, or to those with a large number of variables. Another method of saving memory is to periodically return the value of the stack pointer to its original, power-up, value. This should only be done when all the data already on the stack has been used, and will not be required again.

Certain instructions automatically assume the hardware stack as one of the operands. For instance, PEA (Push Effective Address) pushes the indicated effective address onto the stack.

Syntax : **PEA ea**

Only the effective address needs to be indicated. The address can be given with any of the indirect memory addresing modes, such as (An), with or without an index or displacment from a base address as required.

User and Supervisor Stacks

The 68000 processor has two stacks, but these cannot both be accessed at the same time. The two stacks are the *user stack* and *supervisor stack*. The user stack is that described above, which may be accessed by a program during execution. The supervisor stack is used by the operating system for the processor calculations when executing program code. This stack can only be accessed by a program being executed under supervisor mode. This programming mode is used for the execution of certain instructions, and must be deliberately selected by the programmer. The supervisor mode will be explained and discussed further in Chapter 13.

The main purpose of this programming mode is to prevent the programmer using these special 'privileged' instructions which may disrupt the flow and execution of a user program if used without due care, since they alter flags and pointers. The programming mode currently in use is indicated by the state of bit 13 of the status register. If it is clear, the user mode is being used, if set then the supervisor mode is being used.

Each stack may be accessed only from its own state, although one privileged instruction exists to alter the base address of the user stack pointer, the MOVE USP (User Stack Pointer) instruction is required. This is a privileged instruction and will be discussed in Chapter 13. Both programming modes use A7 as the stack pointer.

Stack errors

There are only two errors associated with the stack, stack overflow and stack underflow. Overflow occurs when the computer tries to place too many items on the stack, such as in an infinite loop which places information on the loop. This type of error, in an otherwise correct program, can generally be avoided by use of stack frames, created with the LINK instruction, as previously described. Underflow occurs when an attempt is made to remove more data from the stack than was originally placed on it.

Subroutines

An important use of the stack is for subroutines. A subroutine is a section of a program which may be repeated several times within the program. In order to save program storage space, the subroutine is given a name, called a 'label', and is defined once at the beginning of the program. Later, during the main program, the subroutine is called by its label instead of being typed again.

Subroutines use the stack to store addresses. When the subroutine is called with a JSR or BSR instruction, the processor stores the contents of the program counter on the stack, and then fetches the address at which the subroutine is stored, and stores it in the program counter. The program execution continues with the subroutine. The end of a subroutine is marked by an RTS (ReTurn from Subroutine) instruction. On execution of this instruction, the processor 'pops' the address stored on the stack into the program counter, and program execution continues with the instruction after the subroutine was called.

e.g. The following instruction, at address $1234 jumps to a subroutine starting at address $5678.

```
JSR  $5678
```

Before the JSR instruction PC was equal to $1237 (the address of the following instruction; remember that the PC is incremented BEFORE the instruction is executed).

After the instruction is executed, PC is equal to $5678. The word value on top of the stack are 1234. Addresses are stored as longwords, sign-extended as necessary.

Subroutines are used to replace sections of code which occur more than once in a program. Some program consist almost entirely of subroutines, called as and when necessary.

Consider, for instance, a games program similar to those in arcades, where a player-controlled monster wanders around a maze avoiding other monsters and collecting objects. A description of the program could be :

1. Draw maze - always the same, so use a subroutine
2. Check if player is pressing a key - subroutine
3. Respond to key press - subroutine
4. Update players position - subroutine
5. Update monsters positions - subroutine
6. Check collision - subroutine
7. If collided, end
8. If not collided, check if player has collected object - subroutine
9. Increase score if necessary - subroutine
10. GOTO instruction 2

This set of subroutines is repeated until the player is dead. The loop described is executed, thus executing each of the subroutines. Although the described program could be written without subroutines, the use of subroutines makes the source program shorter, and much easier to read and understand, making it easier to correct any errors in the program. It is far easier to find the relevant part of the program for correction if the program is divided into clearly labelled subroutines, and is of manageable size.

Passing Parameters

The stack is also used to pass parameters to subroutines. Parameters are variables required by the subroutine for its execution. Since they change and may have virtually any value, they cannot be permanently stored as part of the subroutine, but instead, each time the program is executed, the parameters have to be stored where the subroutine can have access to them. The most usual storage place is on the stack. An alternative is to store the parameters in memory, in a fixed memory location. This is not very satisfactory, since it means that the program is no longer relocatable, an important consideration on machines such as the Apple Macintosh, which do not always load a program into the same addresses in memory each time the program is loaded. If parameters are stored in fixed locations, it is possible that on some occasions, those fixed locations could be part of the program storage area, corrupting the program as it is executed. By storing parameters on the stack these problems are avoided.

The parameters are stored on the stack in the appropriate order immediately before the subroutine is called. The first instruction in the subroutine should be to pull the return address for the subroutine off the top of the stack, and store it in an address register. The rest of the data on the stack can then be pulled off as and when required. The last instruction of the subroutine, before the RTS instruction, should be to restore the return address from the address register to the top of the stack, otherwise an error will occur when the processor cannot find the return address in the place it expects it to be.

Because of its relocatability, this method of passing parameters is preferred to storing parameters in memory, and is also preferred to storing parameters in data registers, since it frees the data registers for calculations using the data.

Chapter Ten Storing Data

One method of storing data for programs has already been discussed, the stack. However, the stack is merely an implementation of a more fundamental method of storing data - the *array*. Arrays are one of two basic methods of storing data. The other is the *linked list*.

Arrays and linked lists are both data storage methods, but the similarity ends there. Whereas the data items in an array follow each other consecutively in memory in order, and are normally the same size, the items of data in a linked list need not follow each other at all, and are generally not in any particular order in memory, and may be of different lengths. The elements in linked lists contain information within themselves indicating the location of the item of data immediately after it in the list, and sometimes the location of the data item immediately before it in the list. It is this information which the computer uses to locate the item of data it requires.

Thus, to locate an item of data in a linked list, the computer has to search through the list until it finds that particular item of data, since the program has no way of telling where in memory the particular item of data may be stored. In an array, however, the items are in a fixed sequence in memory, and may be accessed by adding a positive or negative displacement to the base address of the array, and thus calculating the address at which the data is stored. It can thus be seen that data in an array can be accessed much faster than data in a linked list.

The advantage of linked lists over arrays is shown when modifying, or adding to, the data in an array and in a linked list. To add data to an array all the data in locations lower than the location into which the data is being inserted must be moved down in memory to accommodate the extra data. Adding data to a linked list merely involves altering the data immediately before and after that being altered, so that the address pointers are adjusted to include the new data.

e.g. Consider the following data, arranged first as an array, and secondly as a list.

1. (Array)

Clare.Kelvin.Bill.Richard.José.Teresa.Lyn

2. (List)

Clare-2.Kelvin-3.Bill-4.Richard-5.José-6.Teresa-7.Lyn-1

(The numbers indicate which element in the list is to follow)

To add information to the array about each element would involve moving considerable amounts of data - but to add it to the list involves simply altering the pointers at the ends of the elements, and adding the extra data at the end of the list.

The final list would be:

Clare-8.Kelvin-9.Bill-10.Richard-11.José-12.Teresa-13.Lyn-14.Happy birthday-2.Mac-removal expert-3.Cricketer of the Year-4.Thanks-5.Hi Mãe-6.and the horse-7.Hello-1.

To follow through the list, the element order would be :

1.8.2.9.3.10.4.11.5.12.6.13.14.1....etc.

Taken in order, the list looks like this :

Clare-Happy birthday.Kelvin-Mac removal expert. Bill - Cricketer of the year.Richard-Thanks.José-Hi Mãe.Teresa-and the horse.Lyn-Hello

Linear and Circular Lists

Linked lists may be *linear* or *circular*. Linear linked lists have a definite beginning and end to the list. The first and last elements in circular lists point to each other, so that in effect, the list has no definite end or start.

Arrays are used to store data which needs to be accessed quickly, and which does not need to be altered much throughout the execution of the program. Arrays are used often for *look-up tables*. These are tables of data required by the program. The data is usually such that it would otherwise have to be calculated each time the program required it. Thus, storing it in the form of an array makes access to the data faster, since the time taken to access the data in an array is less than the time required to calculate the value of the data from a formula. In assembly language, there is another advantage, since the number of instructions required to perform the look-up is normally less than that required to perform the calculation. If it requires more instructions to perform the look-up, either you are making a mountain out of a molehill in your programming, or the formula is so simple, no look-up table is required.

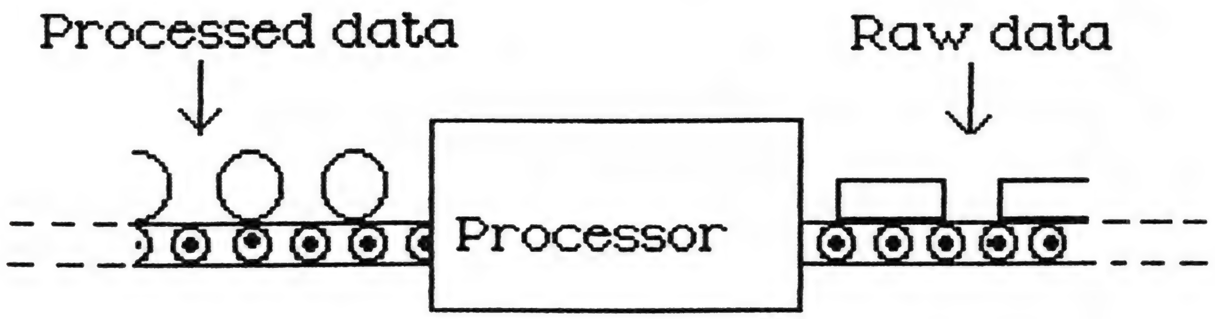
An example of a program requiring a look-up table is one which requires the values of the sines of angles. The table is constructed before the program is run - it may be loaded into memory as part of the main program, either at the end or the beginning. The values for the table are calculated manually or using a BASIC program to calculate them and store them in memory.

Arrays are accessed by adding a displacement to the base address of the array. For a search through an array, the loop merely needs to increment the base address by the number of bytes used by each element. However, this system of accessing the elements means that elements are far easier to find in the array. Most arrays are constructed so that an item is stored at a location whose address number has some relevance to the result of the calculation required to produce, or is related to the data to which it refers. For instance, in the sines of angles table, each sine could be stored at a location whose displacement from the base address is the same as the value of the angle. Thus, to find the sine of 36°, the base address has a displacement of 36 added to it to produce the address at which the sine is stored in the table.

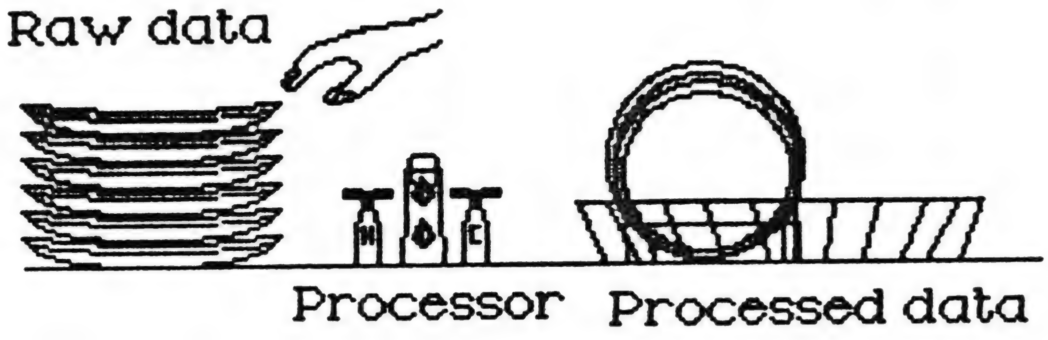
Linked lists are used for data which has to change frequently, or which often has to be added to, since the number of elements affected by an alteration or addition to the list is far less than would be affected by the same change in an array. For instance, the objects held by a player in a simple adventure game could be in a linked list, in which each element contains the data for the object and the address of the next object in the list. When a player drops an object, the element in the list prior to that dropped is altered to point to the element after that dropped, rather than the dropped element. Thus, when the list is accessed to check what the player is holding, the dropped item is ignored.

Arrays and lists are the basic formats for data storage. They are used to form other types of storage, such as stacks, queues and trees. Stacks have already been discussed in Chapter 9, and are generally in the form of an array.

Queues are also generally in the form of arrays. They are similar to stacks, since the data is stored at the end of the queue, and the pointer incremented. The difference between stacks and queues is that, whereas in a stack it is the last item of data placed on the stack is the first item of data removed, in a queue the first item of data stored is the first item removed. Queues can be thought of as conveyor belts of data passing through the processor, with each parcel of data being picked up and processed as it passes, in the order in which the data parcels are placed on the conveyor.



In contrast, stacks are like a pile of dishes, and the 'dishes' of data are taken off in the opposite order to which they were put on the stack, in order not to upset the pile.



Queues and stacks are used according to the way in which the data which is to be stored in them is to be processed. The internal buffers, which hold information which is input to, or output from, the processor are organised as queues, since the information must be dealt with by the processor or the peripheral in the order in which it arrived. If a stack was used, the information would be back to front, and possibly out of order, since the information is not always be sent all at one time.

Queue Pointers

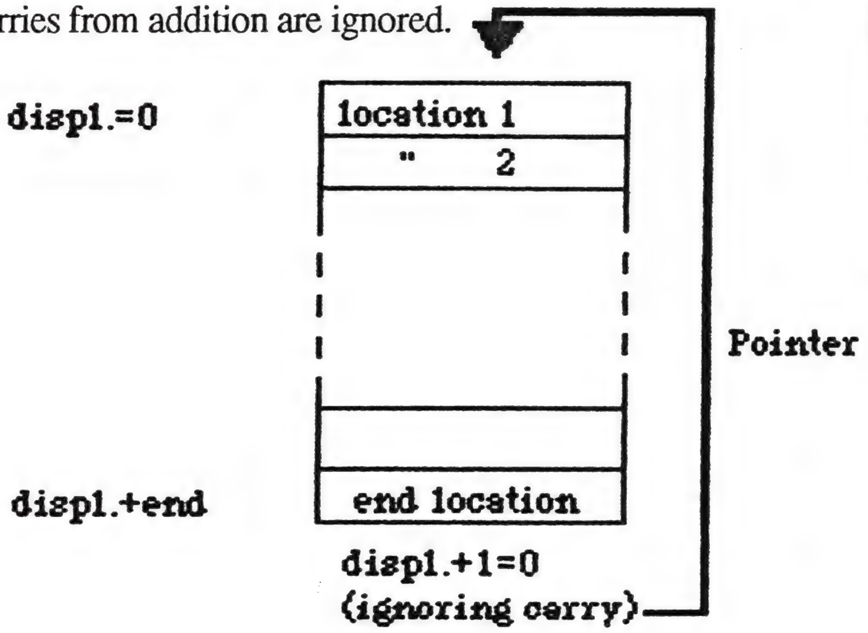
Queues require two pointers, one which points to the *head* of the queue, the next item of data to be processed, and the other which points to the next address at which data can be stored, the end of the queue. The 68000 has no hardware queue, and thus no address registers are specifically designated to be used for this purpose, and any of the eight can be used, although it is always best to leave A7 as the stack pointer and never use it for any other purpose unless it is absolutely essential. The data in the queue is accessed with the (An)+ addressing mode, which automatically updates the pointer to the head of the queue as data is taken off the queue. Data is added with the same addressing mode, to update the pointer for the end of the queue. Queues grow upwards in memory from the head, whereas stacks grow downwards.

Queues are useful in programs which take input from the user, such as word-processing programs. The data received from the keyboard is stored as a queue in memory, before being stored on disc or sent to the printer. Character data passing from the keyboard into a word-processing program goes through three queues. The first is the keyboard buffer, where it waits until it is fetched by the processor as data for the program. It is then stored in two queues, one for data to be sent to the screen and the other for the character data. The second will later be used to provide the data for the disc copy of the file and the print-out of the file.

Queues are used for this type of application for the reasons explained in the note following their use in buffers. All programs which require data to be processed in the order in which it is given use queues. Data read from a disc is stored as a queue, the first item of data read from the disc being the first to be stored in memory, and sometimes the first to be processed when the program is executed. Note that some programs have look-up tables and other data at the beginning of the program, and the program code may not actually start until some way through the data input from disc.

Circular and Linear Queues

Queues may be both circular or linear, regardless of whether they are arrays or lists. The pointers in circular queues are incremented such that when they reach a certain value, they revert back to their original value. Circular array queues form an efficient method of using memory, since the queue is kept within two boundaries. To increment the pointers in this way, the original pointers are stored in address registers, but are not incremented. The displacement of each pointer from the base address is stored in a data register, and these displacment pointers are incremented as necessary after each operation involving the queue. The next entry in the queue array is found by adding together the address and data registers, using the address register+index addressing mode (**An Dn**).The pointers are incremented as a word or longword value as required by the length of the queue. For instance, to implement a queue of length 65536 bytes, only the low-order word of the data register containing the displacement pointer needs to be considered, and any carries from addition are ignored.



For example, for a 65536 byte length queue, if the displacement pointer has the value $FFFF (unsigned) - that is, the last item in the queue, adding $1 to increment the byte length pointer produces the byte length result $0000, with a carry of $1. The carry is discarded, and the queue pointer returns to the base value.

Thus, whenever a pointer falls of the end of the queue, it is automatically reinstated at the beginning. To access a queue of this nature, the pointer is addressed with the *address register indirect with index* addressing mode. The syntax for this addressing mode is (An,Dn.l) where An is the address register containing the base address and Dn is the data register containing the displacement to be added. 'l' signifies the data length being used, which may be word or longword. To implement a queue which is neither $FFFF nor $FFFFFFFF bytes long (i.e. pointers of a exactly a word or a longword), the value of the displacement pointers must be checked each time they are updated, and their contents returned to zero displacement by subtracting the number of bytes in the queue.

e.g. For a queue of 412 bytes, the displacement pointers would be checked after each update. If either pointer contained the value 412 pointing to the end of the queue, then 412 would be subtracted to return the pointer to zero displacement, the head of the queue.

To use a linked list as a queue, tail pointers attached to the data simplify operations to alter or add to the elements in the list.

Although queues are most often used in such a way that they process data in a chronological order, using a linked list structure within the queue enables the queue to process data in other orders, such as in order of priority. For instance, in a system where data is arriving from several sources, and data from one source has priority over data from another source. The data is first checked to determine its source, and is then stored in the queue in its priority position.

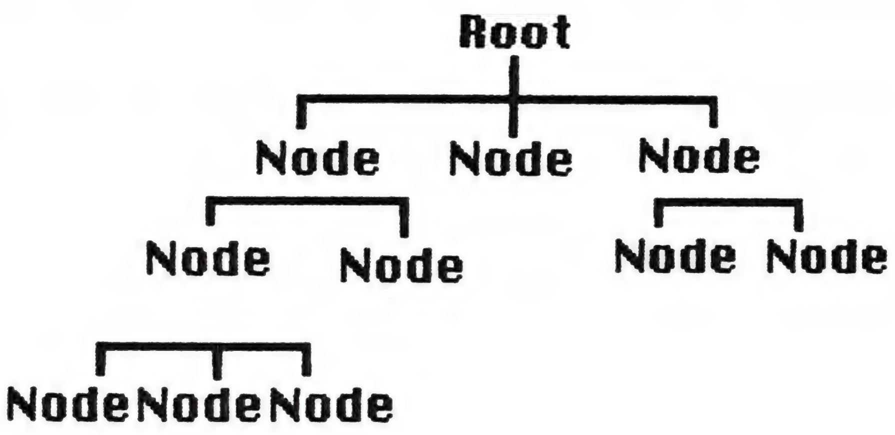
Double-ended Queues

Stacks and queues can be combined to form a structure known as a double-ended queue (dequeue, pronounced 'deck'). This structure allows data to be inserted and removed at both ends - this type of structure requires careful handling to ensure that the correct type of addressing mode is used to access the ends of the queue. At one end, the -(An) mode will be used to add data, whereas at the other end, it will be used to remove data, and vice versa for the (An)+ mode. It is also necessary to be careful not to overwrite data in the dequeue which is still needed by the program.

Other forms of dequeues exist which allow restricted input and output. The first allows data to be inserted at only one end, but removed at both ends, and the latter allows data to be inserted at both ends, but removed at only one end.

The Tree Structure

There is one further data structure which may be superimposed on linked lists. This is the tree structure. Each element may point downwards in memory to several elements, but points upwards to only one element. The first element in the tree is called the root, and this root points to nodes, which in turn point to other nodes and so on down the tree. Within a tree there may also be subtrees, extending from a node of the tree.



Each node in the tree has several pointers - usually four, some of which may be null, containing a terminator. These pointers are known as the left, right, back, and trace pointers. The left and right pointers contain the displacments to the addresses of the pointers to the left and right of the element, the back pointer contains the displacement to the address of the element from which the node leads, and the trace pointer points to the element next in numerical order in the list. The trace pointer allows data to be read in sequence. If a node has no element leading in the direction of a pointer, a terminator value is placed in that pointer. This value is usually 0, informing the program that no element exists in that direction. For instance, the root of a tree has no back pointer.

Although trees are rarely used in user programs, an appreciation of the way in which they are structured is important, since they occur in some system software, such as disc handling software. An example of a tree structure is found in many disc directories, which allow sub-directories to be formed within a main directory and sometimes within other sub-directories. The main directory is the root, and the sub-directories are the nodes. In order to access a program within a sub-directory, it is necessary to go through the main directory and into the sub-directory(s) before arriving at the program. The information which the computer requires may be given as an extension to the program name, sometimes resulting in very long names. For instance, to access a program called 'prog', which is stored in directory 'dir2', which is a sub-directory of directory 'dir1', which is in turn a sub-directory of the root directory 'root', the instruction to load the program could be:

```
                    LOAD   'root.dir1.dir2.prog'
```

N.B. This is NOT a 68000 instruction, but simply an illustration of tree structure in use.

The program name is the last item in the information given to the computer, and the additional information items are distinguished by a separator, in this case a full-stop. Note that the information can only be given in one order, with the root first and progressing logically through the node directories until reaching the destination.

All tree structures are referenced in this way, passing through the tree in logical sequence from one node to the next until the destination element is reached.

A thorough understanding of the data storage structures discussed is required before progressing to the next chapter, since knowledge of data storage and methods of accessing the data stored will be assumed. A knowledge of trees will not be assumed.

Chapter Eleven
Advanced Programming

As a continuation of the data storage techniques explored in the previous chapters, two fundamental program utilities will now be investigated - the *sort* and the *search.*

A sort is a program, or a routine within a program which arranges a set of records into a specified order in memory. For the purposes of this book the records will be assumed to be be stored in memory, although they could be stored on another storage medium, such as disc.

There are three basic forms which a search can take, *insertion* sort, *interchange* sort, and *bubble* sort.

An insertion sort takes the data one item at a time, and places the item in a linked list at its appropriate place in the list. Thus, the data is sorted when all the data has be input. The data may come from the keyboard, from an array in memory, or from a form of storage such as disc. The data for this type of sort is stored most efficiently in a linked list, especially if large amounts of data are being used, since the alteration required to the list when an item is inserted is considerably less than would be required by an array.

The steps for this sort would be:

1. Fetch data item from storage

2. Check through list, comparing data item with each pair of elements.

3. When the pair of elements is such that the data item has a value between those of the two elements, insert the data item at that point, altering the pointer of the first element to indicate the added data, and make the pointer of the inserted data point to the second element of the pair.

4. If not the end of the data, goto 1

The data is ordered in such a way that, to be inserted, the data must be less than the first element and larger than or equal to the second element. If not, the pointers are updated to check the next pair of elements - i.e. the second element of the original pair and its following element.

For example, a circular list such as follows (the numbers after the value indicate the data which follows in the list):

```
1.     $1234...(2)
2.     $3456...(3)
3.     $4567...(4)
4.     $6789...(1)
```

To store a data item of value $4667, the computer would check elements 1 and 2, then 2 and 3, then 3 and 4. The data would then be stored in the next available space (usually at the end of the list data) and the pointer in element 3 adjusted to point to the new data.

The pointer at the end of the data would point back to element 4.

The list would then look like this :

```
1.     $1234...(2)
2.     $3456...(3)
3.     $4567...(5)
4.     $6789...(1)
5.     $4667...(4)
```

Here is a program which will do this, assuming the pointer to the first element in the list to be A6, and the pointer for the data to be A5, loading base pointer in from stack. A2 holds the address where the next element in the list is to be stored.

The program is a routine and is presented in such a way that it can be easily incorporated into any program. Thus, no line numbers are specified, and no particular value is assigned to the pointers.

```
.nextdata
MOVE -(A6), D0                ...get data
MOVEA (SP), A5                ...put list base
                              ...address in A5
.loop
MOVE (A5), D1                 ...put 1st pair
                              ...element in D1
MOVEA -(A5), A4               ...put address of
                              ...next element in
                              ...A4
MOVE (A4), D2                 ...put 2nd pair
                              ...element in D2
CMP D0, D1                    ...compare data to
                              ...1st element
BGE continueloop              ...if bigger,
                              ...continue loop
CMP D0, D2                    ...compare data to
                              ...2nd element
BLT continueloop              ...if less
                              ...continue loop
MOVE A2, (A5)                 ...alter pointer
                              ...of first
                              ...element
MOVE D1, (A2)                 ...put data in
                              ...next loop space
MOVE A4, -(A2)                ...and associated
                              ...address
CMP D0, #$000                 ...end of data?
BEQ end                       ...yes, then end
JMP nextdata                  ...no, get next
                              ...data
.continueloop
MOVEA A4, A5                  ...2nd element =>
                              ...1st for next
                              ...pair
JMP loop                      ...continue loop,
                              ...data not fitted
.end
RTS                           ...end of data,
                              ...thus sorted
```

Note that the stack pointer is NOT addressed with the pre-decrement mode, since its value has to remain constant throughout the sort.

Interchange sorts are slow, and should only performed on small amounts of data. They are worth mentioning for the way in which they function.

The routine sorts through the data by taking each element in turn, as a comparison element and comparing it to each element below it in the data, usually a list. For a list in which the top element is the largest and the bottom element the smallest, the elements being compared are exchanged if the comparison element is smaller than that to which it is being compared. The comparison element then replaces that element in the list, and that element is made the new comparison element and is stored in the position in the list from which the comparison element was taken. This is repeated until every element position in the list has been used as the comparison element. The list is then sorted.

For instance, to sort a list such as this:

1.	$123
2.	$567
3.	$8
4.	$1235
5.	$6780
6.	$6

Element 1 is the first comparison element.
Elements 1 and 2 are compared. 2 is larger than 1, so they are swapped.

1.	$567
2.	$123
3.	$8
4.	$1235
5.	$6780
6.	$6

Element 1 is now the comparison element.
Comparison continues with element 3.
1 is larger than 3, so no swap.
Compare 1 to 4.

4 is larger, so swap.

1.	$1235
2.	$123
3.	$8
4.	$567
5.	$6780
6.	$6

Element 1 is the comparison element.
Comparison continues with element 5
5 is greater than 1, so swap.

1.	$6780
2.	$123
3.	$8
4.	$567
5.	$1235
6.	$6

Element 1 is the comparison element.
Comparison continues with element 6
6 is less than 1, so no swap.
At the end of this pass, the largest element is now on the top of the list.

Element 2 is now used as the comparison element, compared to 3 - 6. After this pass, element 2 will hold the 2nd largest element, and the passes will be repeated for all elements until the last element. The list will then be in order. Each pass can thus be seen to place the largest element in the list below the position of the comparison element into that position.

After sorting the list is :

1.	$6780
2.	$1235
3.	$567
4.	$123
5.	$8
6.	$6

No program example is given here since interchange sorts are virtually never used, being a very inefficient method of sorting. The principles of the sort are included here merely for interest and to show all the different sorts.

Bubble Sorts

Bubble sorts are the most widely used type of sort. This is partly because they can be equally well used with arrays and lists, and because they form a fast and efficient method of sorting data.

A bubble sort takes a consecutive pair of elements in the data and compares them. If the elements are in the opposite order to the order required (the smaller before the larger in a list or array to be ordered with the largest element first, or vice versa), they are exchanged. Since the exhange does not involve any other elements, arrays can be sorted quickly by this method since no great re-shuffle of data in required. In lists, only the appropriate pointers are exchanged.

Each time a pass through the data is made, an indicator is cleared before the pass starts. If a swap is made, this indicator is set. If, at the end of a pass the indicator is set, a swap has occurred, and the data cannot be said to have been sorted. Another pass is therefore made, clearing the indicator before starting. The procedure is continued, making passes through the data and exchanging data as necessary until a pass is made with no swaps and the indicator remains clear.

The routine for a bubble sort is of a form such as follows. The base address of the data array is held in A0. The end address of the array is held in A3. The indicator for a swap is D3. A1 and A2 are used to hold the addresses of the pair of elements being compared. The data of the elements is held in D0 and D1.

```
.pass
MOVEA A0,A1                 ...put base address in
                            ...A1.

.loop
MOVE (A1),D0                ...put first element in
                            ...D0.
MOVE -(A1),D1               ...put second element
                            ...in D1,
                            ...pointer decreased
                            ...before access.
```

```
MOVEA A1,A2              ...store address of
                         ...second element
                         ...formed by
                         ...decrementing
                         ...pointer
                         ...to first element.
CMP D0,D1                ...compare elements.
BGT continueloop         ...if in correct order,
                         ...continue.
MOVE D0,(A1)+            ...swap data elements
                         ...if not in order.
MOVE D1,(A1)             ...note use of
                         ...addressing modes
                         ...to increment pointer
                         ...to access
                         ...both addresses.
MOVE.B #$F, D2           ...set swap indicator.
.continueloop
CMP A2, A3               ...is it the end of the
                         ...pass?
BEQ checkend             ...if yes, jump.
MOVEA A2,A1              ...make 2nd pointer the
                         ...first
JMP loop                 ...and go through loop
                         ...again.
.checkend
CMP D2,#$F               ...has a swap occurred?
BNE end                  ...if not, data is
                         ...sorted, so end.
MOVE.B #$0,D2            ...if yes, reset.
JMP pass                 ...and go through pass
                         ...again.
.end
RTS                      ...end of program.
```

Addressing modes are used to alter the pointers automatically when accessing data, to reduce the number of instructions required. The number of variables required to store addresses in is also reduced to a minimum.

Sorts are generally carried out on data to arrange it into alphabetical order, in which case the data to be compared is ASCII codes. Sorts can be carried out for a variety of reasons, for neatness of presentation, to create a list of data in which a particular item can be easily found, etc. Numeric sorts to order data in numeric order can be carried out to sort data chronologically, if the dates are compared.

Another, often-used, technique is that of searching data for a particular item. There are three basic search techniques, each suitable for a different type of data organisation.

Search everything. This technique is the simplest, and can be the slowest. A search routine of this nature checks every entry in an array, comparing it to the data it is looking for until it finds it. For large amounts of data, the process can take a relatively long time. However, if speed is not of the essence, this is the easiest search to include in a program.

The following is a example of a search routine, searching through an array in memory, whose base address is stored in A0. The data being searched for is in D0.

```
.start
MOVE baseaddress, A0        ...initialise address
                            ...counter.
MOVE data, D0               ...put comparison data
                            ...in D0.
.check
CMP (A0)+, D0               ...compare data.
                            ...Addressing
                            ...mode automatically
                            ...increments pointer.
BEQ end                     ...if equal, end.
JMP check                   ...if not equal, go
                            ...back.
.end
RTS
```

Note that the address counter does not need updating, since an addressing mode is used to automatically increment the counter each time another item of data is checked. The array is assumed to grow upwards in memory from a base address.

This type of sort, although slow, may be used with any data structure. It is worth using on large amounts of data if the effort required to sort the data would be greater than that required to search the data. This sort is also the best type for rapidly-changing, infrequently searched data.

In contrast, a binary search can only be used with data which is in an order, and in an array. A binary search requires two pointers - one pointing to the first element in the data, and the second pointing to the last element in the data. The element of data in the exact middle of these pointers is found - thus the data has to be in an array. This element is compared to the data being searched for. If the element is larger, then the value in top pointer is replaced by the address of the middle element. If the element is smaller, then the address of the middle element replaces the value of the end pointer.(This is for a search through an array going from largest to smallest. Reverse all conditions for an array with its smallest element as the first.) The procedure is repeated - find the middle element between the two pointers, compare and replace - until only one element is left - this will be the data being searched for. The procedure stops at any point if the middle element is the data being searched for.

The following routine illustrates a binary search of an array. The top pointer is in A0, and the last pointer in A1. The data being searched for is in D0.

```
.start
MOVE data, D0                 ...put in data.
MOVE toppointer,A0            ...put in first
                              ...pointer.
CMP (A0),D0                   ...check that it is not
                              ...data being
                              ...searched for.
BEQ found                     ...if it is, end.
MOVE endpointer, A1           ...put in last pointer.
CMP (A1),D0                   ...check that it is not
                              ...data being
                              ...searched for.
BEQ found                     ...if it is, end.
MOVE #$2, D2                  ...put in divisor.
.check
MOVE A0, D1                   ...put in top address.
SUBS A1,D1                    ...subtract bottom
                              ...address.
DIVS D2, D1                   ...divide result by
                              ...two, to obtain
                              ...middle address.
MOVE D1, A2                   ...store address in an
                              ...address
                              ...register.
```

```
CMP D0, (A2)                ...compare with data
                            ...being
                            ...searched for.
BEQ found                   ...if equal, end.
BGT altertoppointer         ...if larger, jump.
MOVE A2, A1                 ...make last
                            ...pointer=middle
                            ...pointer
JMP check                   ...repeat procedure
                            ...with new
                            ...pointers.
.altertoppointer
MOVE A2, A0                 ...make top
                            ...pointer=middle
                            ...pointer
JMP check                   ...repeat procedure
                            ...with new
                            ...pointers.
.found
RTS
```

Hash search

The final search technique is completely different to the previous two. The position of the data address in an address table is calculated from the data itself. An algorithm is performed on the data in order to produce a number. The algorithm is called a hash function and the number produced is called a hash code. The hash code refers to the position of the address of the data in the look-up table of addresses. (The data itself is not usually stored as a table which can be referenced, since the length of the items of data may vary, and there may be more than one item of data for a particular hash code, as will be explained). The purpose of calculating hash codes is to produce a number with which to refer to the data, as an alternative to the value of the data itself, which is smaller than the data value - thus producing a more compact table of data.

The table of data addresses is constructed according to the hash codes of the data - the item with the lowest hash code at one end of the table, and that with the highest at the other end.

Most hash codes are specific to one item of data, but occasionally a hash collision occurs when two items of data produce the same hash code. The number of hash collisions produced is an indication of the quality of the hash function - a good hash function has few collisions.

When a hash collision occurs, all the items of data to which the hash code refers are placed in sequence, starting at the address stored at the hash code position in the address table, and a simple search is used to find the particular item of data required when the hash code is calculated. To indicate that a search is necessary, the first data entry at the address stored at the hash code position in the address table could be set to one of two states, to indicate whether or not a search is necessary, and how many items of data are to be searched.

Another method of storing data and dealing with hash collisions can also be used, which has an advantage of not requiring a separate address table. In this method each data element is of a fixed length, and the high-order longword of the data is an address pointing to any further data with the same hash code. The longword contains a terminator value if there is no more data. Thus, only one item of data is stored in the relevant position in the hash table, and further data relevant to the position is stored at the end of the table, pointed to by the data in the table. This method only works for data of a fixed length, since the positions in the table are each referred to by a displacement from a base address. This displacement is a multiple of the data length. Any data not of the appropriate length will cause the data following it to be stored out of position in the table, and the calculated displacements for the following data will be inaccurate.

e.g. If the displacements are multiples of two, and the data is bytes long. Storing a one byte data item without an addition, null, byte will cause the following data to be stored out of turn.

Position	Address	Data
1.	$1000	$1234
2.	$1002	$2345
3.	$1004	$34
4.	$1005	$2345
5.	$1007	$4567

The example shows items 4 and 5 to be stored at locations $1005 and $1007. However, the displacement for each element is calculated as $1000+(position*2). So the calculated addresses for elements 4 and 5 would be $1006 and $1008 - the two byte data fetched from those positions would not be the data required.

In order to preserve the correct displacements, the data should be stored as follows :

Position	Address	Data
1.	$1000	$1234
2.	$1002	$2345
3.	$1004	$0034
4.	$1006	$2345
5.	$1008	$4567

Element 3 is stored as two bytes.

Storing data longer than the expected length has similar results. Thus, in a table for which displacements from a base address are calculated according to the position in the table, the data should all be of the same length - thus additional data items with the same hash code cannot be stored in memory in consecutive positions to the first item, but must be pointed to and stored elsewhere in memory.

Hash functions, to produce the codes, can be as complex or as simple as you are prepared to spend writing them. Each programmer has their own better-than-everyone-else's hash function, and generally refuses to tell it to anyone but the highest bidder! New and better hash functions for commercial software are continually being sought for by software houses in an effort to cut down the length of time taken to find data in a program, especially now with computers being used extensively in business - speed of execution is often a consideration in decisions over which software package to purchase for a business.

The algorithms for hash functions are many and varied - there are probably as many as there are programmers - but here are a couple of ideas to inspire you to create your own.

Note: Hash functions are performed with the intention of getting a smaller number out of the original number!

1. For a string of characters - multiply the ASCII code of each character by its position in the string, and add all the resulting numbers together, then divide by the length of the string.

2. For a number (or string) - add together all the individual numbers (or ASCII codes) in the number. Find the first prime number (a number which can only be divided completely by itself and one) greater than the resulting total. Find the greatest number of times the total will divide into the prime, and find the remainder left after the division. Combine the two - the remainder and the number of times the total divides - into a number whose low-order part is the remainder, and whose high-order part is the number of times the total divides. This number is the hash code.

The following program calls a subroutine to calculate the hash code. The subroutine itself is not described, so that you may add your own routine to the search program. Also referred to, but not described, is a routine to calculate the address of the data from the hash code. The program assumes that the data is referenced directly (not via an address table), and that the high-order longword contains the address of any other data with the same hash code. For convenience, the data itself is assumed to be one longword.

```
 .start
 MOVE data,D0                  ...put data in D0.
 JSR hashcode                  ...calculate hashcode.
 JSR address                   ...calculate address.
                               ...Assume
                               ...address is stored in
                               ...A0.
 .search
 MOVE.L (A0), D1               ...store extra-data
                               ...address.
 CMP.L D0, -4(A0)              ...compare data.
                               ...Displacement
                               ...sets A0 to address
                               ...of data.
 BEQ found                     ...if equal end.
 CMP.L D1 #$0                  ...compare to
                               ...terminator value.
 BEQ notfound                  ...if equal, data not
                               ...found.
 MOVEA D1,A0                   ...put address of next
                               ...data in A0.
 JMP search                    ...repeat to compare
                               ...next data.
 .found
 RTS
 .notfound
 JSR nowwhat                   ...do whatever,
                               ...re-enter data
                               ...etc.
 RTS
```

The subroutine '*nowwhat*' could be substituted by a routine setting pointers etc. if the data is not found. For instance, the subroutine could print a messsage saying: *'Data not in list - do you want to add it to the list?'*

If the reply is 'yes', then the address pointer, stored in D1, would be substituted by the bottom-of-the-list pointer. The data, preceded by a terminator address pointer would be stored at the bottom of the list, and the bottom-of-the-list pointer updated.

Chapter Twelve Exceptions

So far, all the instructions explored have been general instructions - that is to say, they can be used in the user mode and also in the supervisor mode. The supervisor mode is normally used for operations which involve the operating system, such as exception processing. All the instructions which can be used in the user mode can be used in the supervisor mode, but in addition there are a few instructions which may be used only in the supervisor mode. These are known as privileged instructions. In this way, the system is protected from corruption by user programs, since the privileged instructions capable of corrupting programs can only be used deliberately, so a program is less likely to be able to crash the system than would otherwise be the case.

Privileged instructions are illegal in the user mode, and cause an error if they are used.

The privileged instructions are:

ANDI to SR	...perform immediate AND with ...status register.
EORI to SR	...perform immediate EOR with ...status register.
MOVE to SR	...put a value in the status register.
MOVE USP	...put a new value in the user stack ...pointer.
ORI to SR	...perform immediate OR with status ...register.
RESET	...resets all external devices.
RTE	...return from exception.
STOP	...cause program to wait until a ...reset, exception or trace occurs. ...Then immediate data operand ...following STOP is loaded into SR, ...and PC is incremented so that ...program execution continues.

Any attempt to use a privileged instruction in a user program will cause a TRAP exception to occur.

Exceptions and Exception Processing

A brief review of exceptions and exception processing now follows. A detailed description of the implementation of exception processing in programs is beyond the scope of this book, as it is only used in applications software which may need to deal with errors and interrupts in a different way to the operating system. For the majority of programs, exception processing is unnecessary since the operating system handles the interrupts, and the main purpose of this chapter is to provide an introduction to exceptions, their uses and processing.

Exceptions are events which occur during the execution of a program, which cause the processor stop executing the program and perform a particular task, and then continue to execute the program.

An exception (also called an interrupt) may be external or internal. Internal exceptions are generally caused by program errors,such as division by zero, illegal instructions, trying to access an address which doesn't exist, and certain instructions which cause an exception.

External exception may come from peripherals such as disc drives and printers, bus errors, or external resets.

When an exception occurs, program execution is diverted to an address in memory called the exception vector address. Each different type of exception has its own vector address. At this address is stored a routine which deals with the exception.

However, before the exception is dealt with, the processor stores the status register and program counter onto the supervisor stack so that they may be retrieved later, in order for program execution to continue where it left off.

The address to which execution is diverted depends on the type of exception, and is obtained from the vector number associated with the exception. The vector number is multiplied by four, giving the address of an exception vector which is stored in a table in the supervisor data space. This table of vectors stores the addresses of the routines designed to handle the exceptions. The execution address of the required routine is placed in the progam counter, and the routine is executed. The 68000 vector table is stored in locations $0 to $3FF.

The vectors are stored in numerical order, as follows.

Vector number		Exception
0	-	Reset supervisor stack pointer
1	-	Reset program counter
2	-	Bus error. The address being accessed does not exist.
3	-	Boundary error. Access to a word or longword value was attempted at an odd memory address.
4	-	Illegal instruction. An operation code was encountered which did not correspond to the op code of any 68000 instruction.
5	-	Division by zero. An attempt was made to divide by zero.
6	-	CHK error. CHK ea, Dn compares the twos complement source operand integer to the low-order byte of the data register destination. An exception occurs if the destination contents are less than zero or greater than the source operand.
7	-	TRAPV vector. The TRAPV exception is explained later.
8	-	Privilege violation. A privileged instruction was called from a user mode program.
9	-	TRACE. This exception is explained later.
10	-	A### address error.
11	-	F### address error. Most illegal instruction errors are vectored to vector 4. However, if the illegal code is A### or F###, where the hashes indicate numbers, then the program passes through 10 and 11, respectively.
12 - 14	-	Reserved.
15	-	Unitialized.
16 - 23	-	Reserved.
24	-	Spurious vector.
25 - 31	-	Autovectors.
32 - 47	-	TRAP vectors 0 - 15
48 - 63	-	Reserved.
64 - 255	-	User vectors.

In order to customise the responses to exceptions, the vector addresses stored in the table may be altered to point to your own routines. Most commercial software does this, handling the exception in the best way for the program. For instance, consider a word-processing program. This program would be likely to alter the vector address for an 'out-of-memory' error to point to a routine which would inform the user, and advise them to save the current document.

Exceptions are handled according to a priority hierachy of 0 - 7. Priority 0 exceptions signify no interrupt to program. Priority 7 is the highest priority. This hierachy is needed because the processor may be in the middle of a process when the exception occurs. The priority of the interrupt is then compared with that of the process being executed. If the interrupt priority is higher than that of the process, an exception occurs, otherwise the execption is stored, and executed when the process is finished. The priority of the process is contained in bits 8 - 10 of the status register. Priority 7 execeptions are known are Non-Maskable Interrupts (NMI) because they take priority over all other processes, signifying an event which cannot be set aside until another task is finished.

When the exception is executed, the priority code is stored in bits 8 - 10 of the status register, and the interrupt is acknowledged to wherever it came from. A vector number is obtained for the exception. The program counter and status register are pushed onto the supervisor stack, and the execution address of the exception routine is stored in the progam counter. The exception routine is then executed.

TRAP exceptions

These may be caused either by a program error, or they may be deliberately placed in a program with the TRAP and TRAPV instructions.

```
Syntax:  TRAP #vector ...cause  exception.
         TRAPV        ...cause  exception  only  if  overflow
                      ...flag  set.
```

The TRAP instruction is vectored to one of 15 addresses contained in the vector table between $080 and $0BF. The particular address depends on the low-order nybble of the following operand.

The TRAP exception provides an entry to supervisor mode, and thus allows easy access to operating system routines which may be available only through the supervisor mode. When the TRAP exception (as with all other exceptions) is encountered, the user program counter and status register are stored on the supervisor stack. Control then passes to the routine whose address is contained in the appropriate TRAP vector address, corresponding to the value of the extension operand.

TRACE Exceptions

TRACE exceptions are used to assist correction of programs. The trace mode is accessed by setting bit 15 of the status register, and can only be accessed in supervisor mode, although user programs may be checked in trace mode.

When the program is in trace mode, an exception is generated after each instruction in the program is carried out. The program counter and status register contents are stored on the supervisor stack. Control is then passed to the routine whose address is stored in position 9 of the vector table. This routine will turn off trace mode (by resetting bit 15), and then display the various registers of the 68000, and may also display the value of variables. After the exception routine has been executed, the program counter and status register are restored from the stack, and the next instruction is fetched to be executed. This type of routine is called a debugger, and is essential for correcting assembly language programs since it shows exactly what is happening during the execution of the program. Debuggers vary in their effectiveness and flexibility, and it is best to check carefully the facilities each offers before purchasing.

The essential facilities are :

Display ALL registers - data, address, status, and program counter.
Display instruction being executed.
Display variables on request.
Other facilities whch may be important :
Ability to switch trace mode on and off within program.
Ease of access to the debugger - method of installing in the computer.

Illegal instruction errors

These may pass through one of three vectors (4,10,11), depending on their value. By writing an appropriate routine, these exceptions can be used to extend the instruction set of the 68000. The routine would check the op code which had caused the exception, and execute a corresponding routine. In this way, not only could new instructions be created, but often used routines could be called with a single instruction.

Exception routines are ended with the RTE (ReTurn from Exception) instruction. This instruction pulls the user status register and program counter off the supervisor stack, and resumes execution of the program.

Chapter Thirteen
Binary Coded Decimal

BCD, which stands for *Binary Coded Decimal* is a data format which is used in accounting and other number processing which requires only addition and subtraction of numbers in character format.

It is used because the conversion of decimal numbers to BCD and vice versa is much faster than the conversion of decimal to binary, thus making BCD a more efficient form in which to manipulate numbers.

BCD is represented in a form similar to hexadecimal numbers. Each single figure in the decimal number is converted into a nybble (a four-bit binary number). Thus, one byte represents two decimal numbers. In contrast, binary requires extensive multiplication and division to convert into decimal and to convert from decimal.

Although BCD is not true binary, it looks similar to it and can be stored and accessed in a similar way. It is not necssary to convert decimal to BCD within a program - the computer will do the conversion when the appropriate instructions are given. There are three BCD instructions, for addition, subtraction, and negation.

ABCD - Add Binary Coded Decimal

Syntax: **ABCD Dn, Dn**
or **ABCD -(An),-(An)**
Data length: byte.

The ABCD instruction performs addition with extend. The two low-order bytes of the operands are added in BCD format, and stored in the destination operand.

There are two forms of the instruction:

Data register addition. The two low-order bytes are added and stored.

Memory addition. The instruction is designed for adding multiple bytes - hence the addressing mode used is -(An). If there was a carry over from the addition, the X (eXtend) flag is set and added to the next byte.

SBCD - Subtract Binary Coded Decimal

Syntax: **SBCD Dn, Dn**
or **SBCD -(An),-(An)**
Data length: byte.

The SBCD instruction performs subtraction with extend. The two low-order bytes of the operands are subtracted in BCD format, and stored in the destination operand.

There are two forms of the instruction:

Data register subtraction. The two low-order bytes are subtracted and stored.

Memory subtraction. The instruction is designed for subtracting multiple bytes - hence the addressing mode used is -(An). If there was a carry over from the subtraction, the X (eXtend) flag is set and subtracted from the next byte.

NBCD - Negate BCD

This instruction forms the negative of a BCD number, using the ten's complement method, similar to the two's complement method described earlier. The number is subtracted from a similar length number of 9s - e.g. subtract 234 from 999 and 4321 from 9999 - and then add 1.

Syntax: **NBCD ea**
Data length: byte.

As with the above instructions, the NBCD instruction is designed for multiple byte arithmetic. The X flag is added to the ten's complement, giving the required borrow. Only the Z bit is cleared, to indicate a multiple arithmetic result.

NBCD instructions should begin with the X flag clear and the Z flag set.

BCD arithmetic instructions are used in the same way as the binary instructions described in Chapter 3, with the additional considerations described above. This form of arithmetic manipulation is only required in programs which process large amounts of data - e.g. accounting programs - and only require addition, subtraction and negation. In other programs which only perform a few calculations the speed advantage is outweighed by the considerations required in the use of BCD instructions, and the use of BCD is neither justified, nor necessary.

Chapter Fourteen Programming Style

A program written in assembly language is called a source file. The machine code produced by assembling the source file is called an object file. It is this object file which is executed when the program is run. The object may be saved onto disc as a binary file so that the source file does not have to be loaded and executed to produce the object file each time the program is required. However, the source file should not be discarded. Any errors (and there will be errors!) are far easier to correct in the source file than they are to correct in the object file.

The first rule of assembly language programming:

ALWAYS STORE THE SOURCE FILE ONTO DISC BEFORE CREATING THE OBJECT FILE.

The source file should be saved before creating the source file in case anything is seriously wrong with the object file, causing the system to crash and wipe out all your hard work. Another potential hazard is that the object file may be created in the memory in which you have your source file stored.

Thus, the second rule of assembly language programming:

CHECK WHERE YOU ARE PUTTING THE OBJECT FILE.

Your assembler should have a 'free memory' command indicating what memory is available for programming - where it is and how much there is of it. Make use of this command - it is NOT a good idea to put an object file where it overwrites operating system workspace, assembler workspace, your source file, or any other of the little essentials which may be stored in the memory. Always check that the assembler you purchase has the 'free memory' facility. If it doesn't, don't buy it. It is no use relying on what your manual says is free memory, manuals have a habit of being wrong, and anyway, memory requirements may change depending on what you have done before creating the source file.

Make your source file readable - you are going to have to edit it frequently. Comments placed at the end of a line explaining the function of the command make a program much easier to follow, and when you come to review the program a few months later it makes life much more helpful when you understand how the program works - it is all to easy to find a fantastic program which you wrote two months ago, and not have a clue how it works! Another aid to making programs easy to understand is to have meaningful labels for the various segments of the program - e.g. START for the beginning of the program, not just whatever key you happen to hit at the time, and (if your assembler allows it) labels for addresses instead of the actual number of the address. Make these labels relevant to what is being stored in the address. Labels such as 'Kelvin' and 'Bill' may make your program look very pretty, but 'pagenumber' and 'letteringstyle' etc. are more relevant to the use of the memory location. It is much easier to follow a program if you know what is being stored in the various locations.

All this advice will be seen to have a purpose when you start constructing programs. A program which has no assembly errors when the object file is produced is not nessecarily an error-free program. The assembler only checks the syntax of the file it is translating - any programming errors ('bugs') in the program which are caused by some oversight - e.g. not covering every possible outcome of a calculation - may cause the computer to do some weird (and not necessarily wonderful!) things.

Planning on Paper

The worst of these is 'crash' - almost as horrific as it sounds - in which the computer refuses to do anything, and the only remedy is to switch the computer off and on again. Now you know why you should save all source files onto disc before execution. It is not funny when you lose two or three hours work because the program has got itself stuck in an infinite loop. These, by the way, can be caused when you have a program which is 'jumping' about all over the place. If you don't keep a watchful eye on what's jumping where and when, the program may follow in such a way that control always jumps backwards at a certain point, when it shouldn't. If no conditions exist to stop the jump-back occurring, an infinite loop is set up, and the only remedy for that is to reset the computer.

This sort of error can be avoided by writing neat, tidy, and well-ordered programs. Planning a program on paper - with a flow-chart if necessary - can iron out many mistakes, forcing you to consider what routines the program is going to require, and where to put them in the program. A few minutes spent with a pencil and paper can avoid several hours (usually the small hours of the morning!) spent at the computer, getting greyer by the minute as you try to untangle the spaghetti of well-intended, but badly implemented, routines. A program which is not structured neatly - i.e. doesn't follow logically - can be a nightmare to debug.

The Core Routine

Ideally, the program should be written around a short, simple routine which calls the other routines as and when necessary. The core routine should have a definite beginning and end, and no other routine should intrude upon it. The other routines in the program should be created before the core program, and in the order in which they occur in the program. This is to avoid 'undefined' errors which will occur when the assembler comes across a label which it cannot associate with an address or variable. e.g. If routine X calls routine Y, routine Y should be defined *before* routine X to avoid an error. Here again, calculating which routines are going to be required, and when, can make all the difference in programming and in debugging. An organised program is far easier to write than a disorganised one - if you know exactly what each routine does, the possible results of the routine are far easier to anticipate and allow for.

Relocatable Code

One final point - relocatability. Not all computers load code into memory in the same place each time - notably the Apple Macintosh. In order that programs can always be executed, no matter where they may be in memory, it is best to always use program-counter relative addressing and to represent data by the use of labels. To define a label to have an address value, the label should be defined at the beginning of the program - your assembler manual will have details of the procedure specific to your assembler. The data is then stored in this label location with an instruction such as MOVE #data, label. Likewise, all procedures should be labelled, not referred to with immediate address values. For forward jumps, for which the addresses have been found by substituting the NOP instruction, the appropriate addressing mode is PC relative with displacement. Instead of substituting the dummy value with the NOP address, substitute the difference between the NOP address and the JMP, JSR, or Bcc address from which the jump is made. The difference between the addresses should be proceeded by an asterisk.

e.g. Consider the following code:

```
        :
        JMP   next
        :
        :
        :
        NOP                     next
        :
```

After the first assembly, the NOP instruction is found to be stored at location $13456, and the JMP instruction is at address $12789. The difference between them is $CCD (note: addresses are always given in hex).

The code then becomes:

```
        :
        JMP   *+CCD
        :
        :
        :
        NOP
        :
```

By using these addressing modes and techniques, your program should always execute properly.

Here endeth the lecture. Have fun programming, and remember - it's only your imagination that limits what you can do with a computer!

Appendices

Appendix One
Full 68000 Instruction Set

The full 68000 instruction set is given here, specifying appropriate addressing modes and data lengths, as well as function.

Addressing modes :

1.	**Dn**
2.	**An**
3.	**(An)**
4.	**(An)+**
5.	**-(An)**
6.	**c(An)**
7.	**c(An,Rn)**
8.	**$nnnn**
9.	**$nnnnnn**
10.	**c(PC)**
11.	**c(PC,Rn)**
12.	**#($)nnnn**
13.	**SR**
14.	**CCR**

The addressing modes will be referred to by their numbers. Data lengths will be referred by as .B, .W, .L.

The flags will be given in the order N Z V C X. The condition will be indicated by the following :

-	unaffected
0	reset
1	set
*	not defined
{a}	altered unless destination is an address register
a	altered if appropriate value
(a)	may be altered if appropriate value

The instructions will be given in **bold** type, followed by their function, the addressing modes which may be used, appropriate data lengths, and the flags affected by the instruction.

ABCD	Add binary coded decimal numbers.	
	1.4	
	.B	
	* (a) * a a	
ADD	Add binary numbers	
	1.2.3.4.5.6.7.8.9.10.11	(ea=source)
	3.4.5.6.7.8.9.10.11	(ea=destination)
	.B.W.L	
	a a a a a	
ADDA	Add to address register	
	1.2.3.4.5.6.7.8.9.10.11	
	.W.L	
	- - - - -	
ADDI	Add immediate value	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a a a a a	
ADDQ	Add quickly, number between 1 and 8	
	1.2.3.4.5.6.7.8.9	
	.B.W.L	
	{a} {a} {a} {a} {a}	
ADDX	Add with extend, multiprecision addition	
	1.5	
	.B.W.L	
	a (a) a a a	
AND	Logical AND	
	1.3.4.5.6.7.8.9.10.11.12	(ea=source)
	3.4.5.6.7.8.9	(ea=destination)
	.B.W.L	
	a a 0 0 -	

ANDI	AND with immediate value	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a a 0 0 -	
	13	(ANDI to CCR)
	.B	
	(a) (a) (a) (a) (a)	
	14	(ANDI to SR)
	.W	
	(a) (a) (a) (a) (a)	
ASL	Arithmetic shift left	
	3.4.5.6.7.8.9	
	.B.W.L	
	a a a a a	
ASR	Arithmetic shift right	
	3.4.5.6.7.8.9	
	.B.W.L	
	a a a a a	
BCC	Branch if carry clear	
	Own addressing mode	
	.B.W	
	- - - - -	
BCS	Branch if carry set	
	Own addressing mode	
	.B.W	
	- - - - -	
BEQ	Branch if equal to zero	
	Own addressing mode	
	.B.W	
	- - - - -	
BGT	Branch if greater than	
	Own addressing mode	
	.B.W	
	- - - - -	
BHI	Branch if higher (unsigned)	
	Own addressing mode	
	.B.W	
	- - - - -	

BLE	Branch if less than, or equal Own addressing mode .B.W - - - - -
BLS	Branch if lower or same (unsigned) Own addressing mode .B.W - - - - -
BLT	Branch if less than Own addressing mode .B.W - - - - -
BMI	Branch if minus Own addressing mode .B.W - - - - -
BNE	Branch if not equal Own addressing mode .B.W - - - - -
BPL	Branch if plus Own addressing mode .B.W - - - - -
BRA	Branch always Own addressing mode .B.W - - - - -
BSR	Branch to subroutine Own addressing mode .B.W - - - - -
BVC	Branch if overflow clear Own addressing mode .B.W - - - - -
BVS	Branch if overflow set Overflow .B.W - - - - -

BCHG	Bit test and change	
	1.3.4.5.6.7.8.9	
	.B.L	
	- a - - -	
BCLR	Bit test and clear	
	1.3.4.5.6.7.8.9	
	.B.L	
	- a - - -	
BSET	Bit test and set	
	1.3.4.5.6.7.8.9	
	.B.L	
	- a - - -	
BTST	Bit test	
	1.3.4.5.6.7.8.9.10.11.12	(bit no. in Dn)
	1.3.4.5.6.7.8.9	(bit no. immed)
	.B.L	
	- a - - -	
CHK	Check that destination is less than the source, but greater than zero, else generate an exception.	
	1.3.4.5.6.7.8.9.10.11.12	
	.W	
	* * * * -	
CLR	Clear destination operand	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	0 1 0 0 -	
CMP	Compare destination and source operands	
	1.2.3.4.5.6.7.8.9.10.11.12	
	.B.W.L	
	a a a a -	
CMPA	Compare with an address register	
	1.2.3.4.5.6.7.8.9.10.11.12	
	.W.L	
	a a a a -	
CMPI	Compare with an immediate value	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a a a a -	

CMPM	Compare memory
	4
	.B.W.L
	a a a a -
DBCC	Decrement and branch back, unless carry is clear
	Own addressing mode
	.W
	- - - - -
DBCS	Dec. & branch unless carry set
	Own addressing mode
	.W
	- - - - -
DBEQ	Dec. & branch unless equal
	Own addressing mode
	.W
	- - - - -
DBF	Dec. & branch unless false
	Own addressing mode
	.W
	- - - - -
DBGE	Dec. & branch unless greater than or equal
	Own addressing mode
	.W
	- - - - -
DBGT	Dec. & branch unless greater than
	Own addressing mode
	.W
	- - - - -
DBHI	Dec. & branch unless higher
	Own addressing mode
	.W
	- - - - -
DBLE	Dec. & branch unless less than or equal
	Own addressing mode
	.W
	- - - - -
DBLS	Dec. & branch unless less than or smaller
	Own addressing mode
	.W
	- - - - -

DBLT	Dec. & branch unless less than Own addressing mode .W - - - - -
DBMI	Dec. & branch unless minus Own addressing mode .W - - - - -
DBNE	Dec. & branch unless not equal Own addressing mode .W - - - - -
DBPL	Dec. & branch unless plus Own addressing mode .W - - - - -
DBRA	Dec. & branch always Own addressing mode .W - - - - -
DBT	Dec. & branch unless true Own addressing mode .W - - - - -
DBVC	Dec. & branch unless overflow clear Own addressing mode .W - - - - -
DBVS	Dec. & branch unless overflow set Own addressing mode .W - - - - -
DIVS	Divide signed integers 1.3.4.5.6.7.8.9.10.11.12 .W a a a 0 -
DIVU	Divide unsigned integers 1.3.4.5.6.7.8.9.10.11.12 .W a a a 0 -

EOR	Logical exclusive OR 1.3.4.5.6.7.8.9 .B.W.L a a 0 0 -
EORI	EOR immediate 1.3.4.5.6.7.8.9.13 .B.W.L a a 0 0 - EOR to CCR .B (a) (a) (a) (a) (a) EORI to SR .W (a) (a) (a) (a) (a)
EXG	Exchange registers No addressing mode .W.L - - - - -
EXT	Sign extend a byte to a word, or a word to a longword. No addressing mode .W.L a a 0 0 -
JMP	Jump to another address. 1.3.6.7.8.9.10.11 No data length required. - - - - -
JSR	Jump to a subroutine. 3.6.7.8.9.10.11 No data length required. - - - - -
LEA	Load an effective address into a specified address register. 3.6.7.8.9.10.11 .L - - - - -
LINK	Create a stack frame. No addressing mode No operands - - - - -

LSL	Logical left-shift 3.4.5.6.7.8.9 .B.W.L a a 0 a a
LSR	Logical right-shift 3.4.5.6.7.8.9 .B.W.L a a 0 a a
MOVE	Copy data from the source operand to the destination operand 1.2.3.4.5.6.7.8.9.10.11.12 (ea=source) 1.3.4.5.6.7.8.9 (ea=destination) .B.W.L a a 0 0 -
MOVEA	Move to an address register 1.2.3.4.5.6.7.8.9.10.11.12 .W.L - - - - -
MOVEM	Move memory contents 3.5.6.7.8.9 (mem.=destin.) 3.4.5.6.7.8.9.10.11 (mem.=source) .W.L - - - - -
MOVEQ	Fast-move an immediate data value. No addressing mode (destination is always a data register) .L a a 0 0 -
MOVE to CCR	Copy a value to the CCR. 1.3.4.5.6.7.8.9.10.11.12 .W (a) (a) (a) (a) (a)
MOVE to SR	Copy a value to the status register. (privileged) 1.3.4.5.6.7.8.9.10.11.12 .W (a) (a) (a) (a) (a)
MULS	Multiply signed integers. 1.3.4.5.6.7.8.9.10.11.12 .W a a 0 0 -

MULU	Multiply unsigned integers	
	1.3.4.5.6.7.8.9.10. 11.12	
	.W	
	- - - - -	
NBCD	Negate a binary-coded decimal number	
	.1.3.4.5.6.7.8.9	
	.B	
	* (a) * a a	
NEG	Negate a binary (normal) number	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a a a a a	
NEGX	Binary negation, with extend. Both the operand and the extend bit are subtracted from zero.	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a (a) a a a	
NOP	No operation - is ignored by the processor	
	No addressing mode	
	No operands	
	- - - - -	
NOT	Inverts all the bits in a number, from one to zero, and vice versa	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a a 0 0 -	
OR	Logical OR operation between the two operands.	
	1.3.4.5.6.7.8.9.10.11.12	(ea=source)
	3.4.5.6.7.8.9	(ea=destination)
	.B.W.L	
	a a 0 0 -	
ORI	Perform a logical OR with an immediate value.	
	1.3.4.5.6.7.8.9	
	.B.W.L	
	a a 0 0 -	
ORI to CCR	As above, destination is CCR	
	13	
	.B	
	(a) (a) (a) (a) (a)	

ORI to SR	As above, to SR (privileged)
	14
	.W
	(a) (a) (a) (a) (a)
PEA	Calculate the effective address, and store it on the stack.
	3.6.7.8.9.10.11
	.L
	- - - - -
RESET	Reset all peripherals. (privileged)
	No addressing mode
	No operands
	- - - - -
ROL	Rotate operand bits left
	3.4.5.6.7.8.9
	.B.W.L
	a a 0 a -
ROR	Rotate operand bits right.
	3.4.5.6.7.8.9
	.B.W.L
	a a 0 a -
ROXL	Rotate operand left, copying leftmost bit into extend flag.
	3.4.5.6.7.8.9
	.B.W.L
	a a 0 a a
ROXR	Rotate right, copying rightmost bit into extend flag.
	3.4.5.6.7.8.9
	.B.W.L
	a a 0 a a
RTE	Return from exception. (privileged)
	No addressing mode
	No operands
	a a a a a
RTR	Return from exception, and restore CCR. (privileged)
	No addressing mode
	No operands
	a a a a a

RTS	Return from subroutine. PC and SR are restored from the stack.
	No addressing mode
	No operands
	- - - - -
SBCD	Subtract binary-coded decimal numbers.
	1.5
	.B
	* (a) * a a
SCC	Set byte if carry clear
	1.3.4.5.6.7.8.9
	(same for all Scc instructions)
	.B (same for all Scc instructions)
	- - - - - (same for all Scc instryctions)
SCS	Set byte if carry set
SEQ	Set byte if equal
SF	Set byte if false
SGE	Set byte if greater than or equal to
SGT	Set byte if greater than
SHI	Set byte if higher (unsigned numbers)
SLE	Set byte if less than or equal to
SLS	Set byte if less than or same (unsigned no.)
SLT	Set byte if less than
SMI	Set byte if minus
SNE	Set byte if not equal
SPL	Set byte if plus
ST	Set byte if true
SVC	Set byte if overflow clear
SVS	Set byte if overflow set
STOP	Stop execution of program until a trace, high-priority interrupt or reset exception occurrs. (privileged)
	No addressing mode
	No operands
	a a a a a
SUB	Subtract binary numbers.
	3.4.5.6.7.8.9
	.B.W.L
	a a a a a

SUBA	Subtract from an address register. 1.2.3.4.5.6.7.8.9.10.11.12 .W.L - - - - -
SUBI	Subtract an immediate value. 1.3.4.5.6.7.8.9 .B.W.L a a a a a
SUBQ	Subtract quickly, an immediate value. 1.2.3.4.5.6.7.8.9 .B.W.L a a a a a
SUBX	Subtract with extend. 1.5 .B.W.L a (a) a a a
SWAP	Exchange the low and high order words of a data register. No addressing mode. No operands a a 0 0 -
TAS	Test and set the msb of a byte operand 1.3.4.5.6.7.8.9 .B a a 0 0 -
TRAP	Cause a TRAP exception. No addressing mode No operands - - - - -
TRAPV	Cause TRAP exception if the overflow flag is set. No addressing mode No operands - - - - -
TST	Compare an operand to zero. 1.3.4.5.6.7.8.9 .B.W.L a a 0 0 -
UNLK	Destroy stack frame created with LINK. No addressing mode No operands - - - - -

Appendix Two Differences between the 68000, 68008, & 68010

The three processors are fairly similar, being based on the 68000 processor. The 68008 processor is a less powerful version, while the 68010 is more powerful than the 68000.

A short description of the 68000 is given below, followed by details of the differences in the 68008 and 68010.

68000

The 68000 has eight 32-bit data registers, and eight 32-bit address registers. The addressing range is 16 Mbytes, since only the low-order 24 bits of the 32-bit address bus are used by the processor. The remaining eight bits are reserved for further expansions in later versions of the chip.

The input/output area is memory-mapped. There are 56 instructions and 14 addressing modes.

Operations may be performed on bit, BCD (4 bit), 8-bit, 16-bit and 32-bit data.

68008

The instruction set is identical to that of the 68000. The data bus is eight bits, not 16 bits, thus most instructions take twice as long to execute as on the 68000, since the data must to sent in two stages. The addressing range is only 1 Mbyte, since only the low-order 20 bits of the address bus are accessed. Addresses above 1 Mbyte are automatically shortened by ignoring the leftmost bit(s) of the number.

There are only two interrupt pins on the chip, instead of three. Thus the 68008 only recognizes four interrupt priority codes (0,2,5,7), compared to the eight (0-7) recognized by the 68000.

68010

The instruction set for the 68010 contains all the instructions of that of the 68000, but also contains a few additional instructions to take advantage of the enhanced capabilities of the 68010. These instructions are MOVEC, MOVES, MOVE from CCR. MOVE from SR is a privileged instruction on the 68010.

MOVEC - MOVE Control Register - This is a privileged instruction which is used to move data to or from the control and function registers, which are implemented only on the 68010. Such a control register is the VSR - vector base register. This register is used, together with an offset, to create a exception vector address. The function registers are two 3-bit registers, the source code register and destination code register. MOVEC sets the registers.

MOVES is used to read or write to locations in areas which cannot normally be written to by a programmer. These areas included the supervisor program area, the user program area, and the user data area. The area accessed is specified by the values placed in the function codes registers. This instruction is priveliged.

The 68010 can access virtual memory, i.e. a data storage area which is not ROM or RAM, such as a disc. However, the processor takes data from the storage area as though it were from memory. This cannot be done by the other processors in the series, and is achieved by suspending normal data retrieval while the data is fetched from the virtual memory, and is placed in physical memory. Normal data retieval has to be suspended, since it takes longer for data to be fetched from virtual memory than is required to fetch data from physical memory, and the fetch-execute cycle would be disrupted if it where not suspended. With this method, large quantities of data can be accessed by the processor, without the need for vast amounts of physical memory to exist within the computer.

An additional feature of the 68010 is that the vector address table may be moved to another area of memory.

Certain instructions, such as those for division and multiplication, execute more quickly on the 68010 than they do on the 68000.

Appendix Three Machine Dependent Routines

The routines discussed in this appendix include such things as printing to both the screen and printer, reading and writing to disc and reading keyboard input. In fact, they are all tasks relating to input and output; communication with the rest of the world.

Unfortunately, these routines vary with each computer, since each manufacturer tries to exploit the capabilities of the machine to the full. Since each machine is different, these routines are also different. This makes life a little awkward for programmers, since a program written on one machine, such as an Apple Macintosh, will not run on another machine, such as an Atari ST, without considerable alteration to the sections of the program which deal with input and output.

All computers have these routines permanently stored in memory, to make programming easier. The way in which the routines are used varies according to the computer, but in general, the data required by the routine is stored in memory, and the routine is then called with a JSR, or similar, command. The exact manner in which the data is stored depends on the machine and the assembler being used. Your assembler manual should contain details of how to access these routines, and it is also worth checking magazines which may sometimes include programs for, and articles about, your machine. Studying these articles can often give valuable hints on programming techniques, as well as giving details of the use of standard routines.

As stated before, data required by the routines is stored in memory prior to its use by the processor. The way in which data is stored varies according to each machine. Some require that the address at which the data is stored is given in a particular register, some require the data to be held on top of the stack and some (such as the Apple Macintosh) require the data to defined in the program, signified by a particular label - e.g. ICON being the data for the details of an icon to be displayed in a menu. The routines assume that the data has been stored in a particular area, or under a particular label. The data is fetched from this area, and is not checked by the processor - thus it is up to the programmer to ensure that the data is indicated correctly in the program, otherwise strange results may occur. In the case of a machine which requires the data address to be specified in a register, or a particular label to be used to identify the data, if the data is not represented correctly (e.g. illegal address in the register, incorrect label) an error will occur.

It is possible to create your own routines for input and output, without using the standard routines, but this is not a particularily useful idea unless you have special requirements - such as non-standard screen graphics, when creating a game, or if you are creating a new disc operating system.

There are areas of RAM in the computer which are reserved for input and output. These are mainly buffers, in the case of output and input to printers and discs, but there is also an area of memory set aside for the screen. In this memory, each location refers to one pixel on the screen. This is called a memory map of the screen. In order to change the state of one pixel on the screen, the corresponding location in the memory map is changed appropriately. The size of the location (bit, byte etc) depends on the type of screen display. A colour display requires more memory for each pixel than a black and white display, since the location has to hold one of several values according to the colour of the pixel. A pixel on a black and white display needs to hold only one of two values, and thus can be represented by a bit, with 0 corresponding to black and 1 corresponding to white (or vice versa). Since the screen display is made up of combinations of pixels, to create a picture on the screen is merely (!) a matter of setting the locations to their required values.

The following program is a simple example, which inverts the state of each pixel on the screen. This particular program is written for the Apple Macintosh, whose screen memory map starts at $07A700, and takes 5472 bytes.

```
00010     SCREEN    EQU        $07A700          Screen map
                                                starts at $7A700.
00020               ORG        $10000           Program starts
                                                at $10000.
00030     START     MOVE.W   #100-1,D2          Begin program
                                                execution. Set
                                                loop count for
                                                100 - the screen
                                                will be inverted
                                                100 times.
00040     .1        LEA      SCREEN,A0          Store first
                                                screen
                                                address in A0.
```

```
00050               MOVE.W   #5472-1,D1       Store loop count
                                              for number of
                                              bytes to be
                                              inverted - 5472.
00060               MOVE.L   #$FFFFFFFF,D0    Value to EOR
                                              with existing
                                              contents of
                                              memory, to
                                              invert them.
00070      .2       EOR.L    D0,(A0)+         Invert memory,
                                              and increment
                                              address for next
                                              screen location
                                              to be inverted.
00080               MOVE.W   #2000-1,D3       Delay loop - so
                                              inversion is not
                                              too rapid.
00090      .3       DBRA     D3,.3            Delay loop - this
                                              repeats 2000
                                              times.
00100               DBRA     D1,.2            Invert next byte,
                                              unless all screen
                                              done (count
                                              completed).
00110               DBRA     D2,.1            If all screen
                                              done, then invert
                                              again, unless
                                              already done 100
                                              times.
00120               RTS                       If done 100
                                              times, then end.
00130               END
```

Note the use of a delay loop. When using a machine with a fast processor such as a 68000, delay loops are often necessary to slow down the program so that it is possible to see what is happening. If the delay loop were not included, (lines 90 and 100) then the screen would appear to be flashing black and white, with each screen colour change appearing instantaneous.

For more complex displays, individually setting or clearing the pixels is not the easiest task in the world, which is why it is generally better to use the machine routines for simple tasks, such as printing text. All non-screen operations are best handled by the standard machine routines, unless you are a masochist or very security-conscious. A favourite technique for protecting programs is to store the main program onto disc using a non-standard write routine, and then have a start-up program which alters the disc-read routine correspondingly, to read the main program. The main program cannot then be loaded without having previously loaded the start-up program, which usually contains other forms of protection preventing inspection of the program, and thus the alterations to the disc read/write routines.

Details of these routines, and their location in memory, and syntax of use, should be freely available, either in books specific to your machine, or in your asembler manuals. Another source of information is the manufacturer of your machine. If you write a nice, polite letter to them, and then wait, they should reply, either with a list of books containing the details you require, or with a request for some money before they release the details you require, or (if you're really lucky) with the information itself! It is worth checking bookstores and catalogues before writing to the manufacturers, since their information tends to be rather over-technical, and it will usually require a fair amount of patience and searching to locate the particular information you require. Books available generally about your machine should be easier to sort through for information.

Appendix Four
Glossary of Terms

68000	16/32 bit processor made by Motorola. Used in many new computers.
Absolute addressing	An addressing mode in which the address to be accessed is given as an extension of the instruction.
Address	Number uniquely identifying a particular location.
Address bus	A set of wires through which are passed the addresses of locations to be accessed.
Address register	Longword register used to store addresses to access locations.
Addressing modes	Different syntaxes for data representation, so that each instruction accepts data from various sources, without using a different instruction.
Addressing range	Number of locations which can be accessed by the processor.
ALU	Arithmetic and Logic Unit. Where all arithmetic calculations are executed.
Array	Structure for storing data, with all the data stored in consecutive locations.

ASCII	American Standard Code for Information Interchange. All computers use this code system to represent characters in memory. Each character is represented by a byte.
Assembler	A program which translates a source file in assembly language into an object file in machine code.
Assembly language	A low-level language, with each instruction representing one machine code instruction. The source file created in assembly language is compiled into machine code. Assembly language provides a neat and simple way to program in machine code.
BCD	Binary Coded Decimal. A form of number representation where each digit of the decimal number is replaced by a binary nybble. Conversion between the two number bases is fast and simple. Used in accounting etc.
Binary	Base two. Numbers are represented using 1 and 0.
Binary search	A search which looks for an element in an ordered array by continually dividing the array until the required element is found.
Bit	A Binary digIT. Has value 1 or 0.
Boolean logic	Logic operations performed on binary numbers.
Branch	A point in a program where execution may continue with one of two routes, depending on certain conditions.

Bubble sort	A sort technique in which passes are made through the data, comparing adjacent data elements and swapping if necessary until the data is sorted into the required order.
Bug	An error in a program. The term originates from the days of computers the size of buildings, when insects crawled into the circuitry of the computer and became fried by the heat, sometimes causing a short-circuit.
Byte	Eight bits. The standard length of a location in memory.

A byte consists of eight bits, as below

128	64	32	16	8	4	2	1

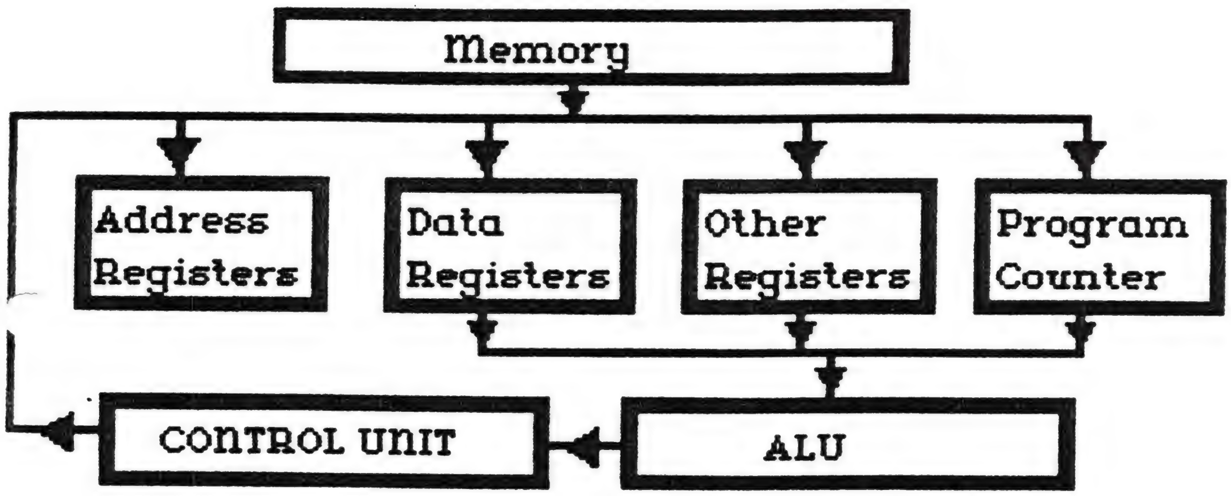
Above each bit is its decimal value

Thus

$$
\begin{aligned}
00001011 &= 0 \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 \\
&+ 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\
&= 11
\end{aligned}
$$

CCR	Condition codes register. This contains flags indicating the result of an operation.
Circular list	A list in which the head is indicated by the tail, and vice versa.

Compiler	A program which converts a source file into an object file, (machine code) prior to the execution of the program. The object file is used to execute the program. The source file is not required for execution.
Compiled language	A language in which the programs are translated with a compiler, rather than an interpreter .
Control unit	A part of the CPU which oversees the passage of data through the CPU, and generally organizes the operation of the CPU.
CPU	Central processing unit. This area of the processor performs all calculations, and executes programs.



Central Processing Unit

Data	Information stored in the computer, in a form understood by the computer.

Data bus	A set of wires which carry data between the CPU and memory.
Data register	A location in the CPU used to temporarily store data required for operations.
Debugger	A program used to locate errors in other programs.
Dequeue	Double-ended queue. Data may be removed or added at either end of the queue.
Effective address	An operand of an instruction.
Exception	An event which causes the processor to stop executing the user program and start executing a supervisor program to deal with the event.
Fetch-execute cycle	A procedure which is repeated by the processor continuously while the computer is switched on. The processor fetches an instruction, executes it, and this process is repeated until the computer is switched off. The processor may be executing either a user program, or the operating system program, performing such tasks as updating screen display, checking the keyboard for input, etc.
FIFO	First in, first out. A data structure used in queues, where the first item of data placed in the queue is the first to be removed and processed.

Flag	Any of the bits in the status register. The state of the flag indicates the result of the previous operation. e.g. If the zero flag is set, the previous result was zero.
Floating point	A format for representing non-integer numbers, where the number is represented by integers and several digits after a decimal point.
Frame	See 'Stack frame'
Hash code	A code produced by a hash function, indicating the position of an item of data in an array.
Hash collision	A hash collision occurs when two different items of data produce the same hash code.
Hash function	An algorithm to produce a hash code from an item of data. A hash function should be constructed to give a different hash code for each item of data. This is rarely acheived, and the quality of a hash function is determined by the number of hash collisions it produces - a good function has few collisions.
Hash search	A procedure for searching for a specified item of data in an array. A hash code is produced from the data. This hash code indicates the position of the data in the array. If a hash collision occurs, a linear search is performed at that position.
Head	The beginning of a data structure.
Hexadecimal	Numbers represented in base sixteen.

High-order bit	The leftmost bit of an item of data.
Immediate value	An item of data given directly as an operand to an instruction. i.e. not as a value stored in a register or memory location.
Index register	A register which stores a displacement value to be added to another value to form an address.
Indirect addressing	The address to be accessed is stored in an address register, or is calculated by adding a displacement to a value contained in a register.
Insertion sort	A sort procedure where an item of data is compared to each item in the rest of the list. If the element in the list is larger, it replaces the comparison element, and the sort continues with that element. This is repeated for each element from the top downwards.
Interchange sort	A sort procedure where pairs of data are compared and the elements exchanged if necessary.
Interrupt	An exception caused by an external device. e.g. printer.
Jump	An instruction which causes program execution to be passed to another point in the program, instead of passing to the next instruction.

/

Label	A name given to a memory location. Labels are used to make programs easier to read, and also as a reference for loops and backward jumps, instead of calculating an address for the jump to pass to.
Least significant bit	The rightmost bit of an item of data.
LIFO	Last in, first out. A data structure used in stacks, where the last item of data placed on the stack is the first removed and processed.
Linear list	A list in which the last element indicates that there are no more elements in the list.
Linked list	A data structure in which each element points to the next element.
Longword	A 32-bit data item.
Loop	A section of a program which is repeated until either a condition is satisfied, or for a specified number of passes.
Low-level language	A computer language where each instruction corresponds to one machine code instruction.
Low-order bit	The rightmost bit of an item of data.
Machine code	The fundamental language of the computer, where instructions are given as numbers.
Memory	The data storage area in a computer. The data stored is in the form of numbers, and may be a program, or data for a program.

Mnemonic	An assembly language instruction.
Monitor	A program used to examine the execution of a program. It is often used in conjunction with a debugger to correct programs.
Most-significant bit	The leftmost bit of an item of data.
NOP	No operation. The processor ignores this instruction, and so it is frequently used to mark locations within programs, to determine addresses for jumps.
Object file	The machine code produced when a source file is compiled.
Op code	The number of an instruction in machine code.
Operand	An item of data required by an instruction.
Operating system	A program which controls the running of the computer.
Patch	A small program used to enhance or correct another program.
PC	Program counter. This register contains the address of the next instruction to be executed. It is automatically updated when an instruction is fetched for execution.
Pointer	An indicator of the address of an element in a data structure. The pointer may be a register - such as the stack pointer, or it may be part of an element in a data structure such as a list, where each data element points to the next element in the list.

Privileged instruction	An instruction which may be executed only in supervisor mode.
Processor	The chip within a computer which executes all the instructions in programs, and carries out general functions for the running of the computer.
Program	A set of instructions which are executed by the processor, in order.
Queue	A data structure, where data data is stored before being processed. The first item placed on the stack is the first to be processed.
RAM	Random access memory. The memory in a computer where data and user programs are stored. RAM is only temporary storage, and loses all the data stored in it when power is switched off.
ROM	Read only memory. No data may be stored in this type of memory. The programs stored in ROM are permanent - they are not lost when power is switched off. The programs stored here are those essential to the computer - such as the start-up procedure. They may not be altered.
Rotate	An operation performed on an item of data, where all the bits are rotated left or right -no bits are lost.
Search	A procedure for locating an item of data within a list or array.

Shift	An operation on an item of data where the bits are shifted left or right - the bits shifted off the end are lost.
Sign bit	The leftmost bit in two's complement arithmetic, indicating whether a number is positive or negative. 0 - positive, 1 - negative.
Signed integer	An integer represented in two's complement arithmetic, where the leftmost bit is a sign bit, not part of the number.
Sort	A procedure for arranging data into an order.
Source file	A program written in assembly language.
SR	Status register. A register containing flags which indicate the state of the computer and the operations being carried out by the processor.
Stack	An area in RAM set aside for temporary storage of data being used by operations.
Stack frame	An area of the stack set aside for storage of data which is destroyed when no longer required. This stops the stack from continuously growing.
System byte	The upper part of the status register which contains information about the state of the computer systsem, such as which mode of program is being executed - supervisor or mode.
Tail	The end of a data structure.

Test	An operation carried out on a bit or byte to determine whether it is zero.
Trace	An exception which occurs after every instruction is executed, if the trace bit is set. This is used by monitors and debuggers to step through a program.
Trap	An exception which occurs when a TRAP instruction is encountered in a program - used to enter supervisor mode.
Tree	A data structure where each element may point to more than one other element.
Unsigned integer	An integer stored in the computer in normal representation. i.e. the leftmost bit is part of the number, not a sign bit.
Vector	A location in memory to which program control passes when an exception is encountered. There are different vectors for each exception. The vector location contains an address to which the program control next passes. At this address is stored a program to deal with the exception.
Word	A 16-bit data item.